CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Human interfaces to structured documents
Authors	Flynn, Peter
Publication date	2014
Original Citation	Flynn, P. 2014. Human interfaces to structured documents. PhD Thesis, University College Cork.
Type of publication	Doctoral thesis
Rights	© 2014, Peter Flynn. - http://creativecommons.org/licenses/by-nc-nd/3.0/
Download date	2026-11-16 19:11:27
Item downloaded from	https://hdl.handle.net/10468/1690



UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Human Interfaces to Structured Documents

The usability of software for authoring and editing

Peter Flynn
MA, FICS
102229277

Thesis submitted for the degree of Doctor of Philosophy



NATIONAL UNIVERSITY OF IRELAND, CORK

SCHOOL OF APPLIED PSYCHOLOGY

August 2014

Head of School: Dr John McCarthy

Supervisors: Dr Jurek Kirakowski
Dr John McCarthy

This document is available from the Cork Open Research Archive (CORA) at
http://cora.ucc.ie.

Contents

List of Figures v
List of Tables ix
Abstract xiii
Acknowledgements xv
Extended abstract xvii
Terminology xxiii

1 Background, scope, and methodology 1
1.1 Background 3
1.1.1 Evidence for structure in writing 4
1.1.2 The use of structure 8
1.1.3 The document model 16
1.1.4 Markup theory and practice 24
1.1.5 Representation 35
1.1.6 Tag abuse 37
1.1.7 Definitions 39
1.2 Scope of the research 42
1.2.1 The nature of the problem 42
1.2.2 Constraints 43
1.3 Methodologies and tools 44
1.3.1 Methodological development 45
1.3.2 Hybrid methodology 45
1.3.3 Persona analysis 47
1.3.4 Methodology and selection 51
1.3.5 Testing 53

2 Research into the writing and editing of structured documents 55
2.1 Usability 57
2.1.1 Development 57
2.1.2 User-Centred Design 59
2.1.3 User experience and Users' Experiences 65
2.1.4 Affordances 66
2.1.5 Testing and measuring 70
2.2 Editing software 73
2.2.1 Historical perspective 73
2.2.2 Usability and structured editing software 85
2.2.3 Design processes 89
2.3 Structured documents 91
2.3.1 Technical writing 92
2.3.2 Document type design 93
2.3.3 Typographic design 95
2.3.4 Interface guidelines 98
2.3.5 Synchronous typographic editing 99
2.4 Summary 102

3 Data collection and analysis **103**
3.1 Expert survey . . . 107
3.1.1 Objectives and design . . . 107
3.1.2 Sample selection and administration . . . 109
3.1.3 Analysis . . . 111
3.1.4 Conclusions . . . 131
3.2 Software analysis . . . 133
3.2.1 Background . . . 134
3.2.2 Software selection . . . 135
3.2.3 Objectives . . . 137
3.2.4 Methodology . . . 137
3.2.5 2005 results . . . 139
3.2.6 2009 results . . . 148
3.3 Requests analysis . . . 161
3.3.1 Objective and methodology . . . 162
3.3.2 Procedure . . . 163
3.3.3 Refining the sample . . . 166
3.3.4 Categorisation . . . 167
3.3.5 Analysis . . . 168
3.4 User survey . . . 176
3.4.1 Questionnaire construction . . . 176
3.4.2 Survey administration and processing . . . 185
3.4.3 Analysis and results . . . 189
3.5 Conclusions . . . 206

4 Modelling and testing **209**
4.1 What have we measured? . . . 210
4.1.1 Principal features and functions . . . 212
4.2 Methodology for building and testing . . . 217
4.2.1 Personas revisited . . . 220
4.3 Semantics of the selected functions . . . 222
4.3.1 Non-menu controls (keyboard shortcuts) . . . 226
4.3.2 Creating and opening documents . . . 244
4.3.3 Insertion of new material . . . 246
4.3.4 Formatting controls . . . 253
4.3.5 Block moves . . . 260
4.3.6 Referencing . . . 269
4.3.7 External insertion . . . 278
4.3.8 Editorial assistance . . . 279
4.4 Building the model . . . 281
4.5 Testing . . . 283
4.5.1 Test harness . . . 284
4.5.2 Test schedule . . . 285
4.5.3 Rubric . . . 285
4.5.4 Expected responses . . . 287
4.5.5 Administration . . . 290

5 Results **297**
5.1 Data processing . . . 298
5.1.1 Computation . . . 300
5.1.2 Analysis . . . 303
5.2 Individual test results . . . 307
5.2.1 Test 1: Create a new Article Journal document . . . 307
5.2.2 Test 2: Add a new paragraph after the current one . . . 309
5.2.3 Test 3: Split a paragraph into two . . . 310
5.2.4 Test 4: Join a paragraph to the preceding one . . . 312
5.2.5 Test 5: Join a paragraph to the following one . . . 313
5.2.6 Test 6: Add a new section to the article . . . 315
5.2.7 Test 7: Adding a new list . . . 317
5.2.8 Test 8: Move a block of text from one place to another . . . 319
5.2.9 Test 9: Highlighting a trade name (product name) . . . 321
5.2.10 Test 10: Add a cross-reference to another section . . . 323
5.2.11 Test 11: Give a citation to an article you need to refer to . . . 325
5.2.12 Test 12: Insert a fragment of another document . . . 326
5.3 Divergences from patterns in tested behaviour . . . 329
5.4 Comments and discussions . . . 332
5.4.1 Informal responses . . . 332
5.4.2 Criticisms . . . 334

6 Summary and conclusions **337**
6.1 Summary of findings . . . 338
6.2 Overall conclusions . . . 341
6.3 Implementation and wider applicability . . . 341
6.3.1 Target Markup Adoption (TMA) . . . 342
6.3.2 Smart Insertion (SI) . . . 344
6.3.3 Moving blocks of text . . . 344
6.3.4 Backspacing and deleting . . . 347
6.3.5 Feedback when inserting new material . . . 349
6.3.6 Adding space . . . 350
6.3.7 The toolbar . . . 351
6.3.8 Editing . . . 352
6.3.9 Unstructured editing . . . 353
6.3.10 Markup characters . . . 354
6.3.11 DTD/Schema compilation and use of stylesheets . . . 355
6.3.12 Appearances . . . 357
6.3.13 Default option . . . 358
6.3.14 Mathematics . . . 359
6.3.15 Wider implications . . . 360
6.4 Recent changes and further work . . . 364
6.4.1 Historical note . . . 364
6.4.2 Recent changes . . . 365
6.4.3 Further work . . . 367

A Expert survey **371**

B Requests analysis **375**
B.1 Shell script to do a search 375
B.2 XSLT script to extract the thread link 378

C User survey **381**
C.1 Questionnaire form 381
C.2 Routines used in data processing 392
C.2.1 The `makedata.awk` script 392
C.2.2 The `readdata.xfr` control file 394
C.2.3 The `reorg.xfr` control file 395
C.2.4 Matching patterns with test data 395
C.2.5 Dataset creation 396

D Tester recruitment and introductory procedure **399**
D.1 Explanatory notes read to testers 404
D.2 Test recording sheets and scripts 408

Bibliography **425**

List of Figures

1.1 Clay tablet calendar, Gezer, *c*.925BC 5
1.2 Roman military report from Vindolanda (tablet 155) 6
1.3 Structure in early legal documents 7
1.4 Structure in early scientific books 9
1.5 Text layers and hierarchy 11
1.6 Orphaned paragraph 13
1.7 Components of a document expressed as 'containers' 19
1.8 Components of a document expressed as a tree 21
1.9 Relationship of markup categories 26
1.10 Example of XML: the recipe definition in use 30
1.11 Required elements being added automatically by an editor 31
1.12 A skeleton document created in a LaTeX editor 34
1.13 Initial persona categorisation 48
1.14 Personas 50

2.1 Some milestones in the evolution of text editing, processing, and markup 73
2.2 Example of procedural markup (*TROFF* manual, 1976) 76
2.3 Implicit markup by type size and weight 96
2.4 Typographic design using indistinguishable styles 97

3.1 Expert survey: Q.1 Participant affiliations 110
3.2 Expert survey: Q.1 Background variables: Operating systems experience 111
3.3 Expert survey: Q.1 Background variables: Structured systems experience 112
3.4 Expert survey: Q.1 Background variables: Years of experience . . . 112
3.5 Expert survey: Q.1 Background variables: Operating systems experience 112
3.6 Expert survey: Q.2 What criteria do you use when selecting editing software? 113
3.7 Expert survey: Q.2 What would be your personal choice of product [. . .] for a) writing and b) editing? 116
3.8 Expert survey: Q.3 Are there products you would prefer to use but cannot (for reasons of price, licensing, availability, platforms, etc)? . 122
3.9 Expert survey: Q.4 Can you identify up to three things your preferred software does which you consider marks it out as specially useful? 122
3.10 Expert survey: Q.5 Can you identify up to three things about any of the structured-document software that you have tested or used which you consider are particularly poor? 125
3.11 Expert survey: Q.6 Are there any other facilities in which structured-document software you have experience of is specially lacking? 127

3.12 Expert survey: Q.7 While using a structured-document software product, have you ever failed to identify how to do something the product was actually capable of? 129
3.13 Software analysis: Discrepancies between 2005 and 2009 measurements 152
3.14 Software analysis: summary of editing character behaviour 159
3.15 Original requests in discussion forums for editing software for structured documents, 1990–2006 (unfiltered) 167
3.16 Encoding the data 188
3.17 User survey: distribution of responses by Experience and use of Operating System against Organisation and Employment 190
3.18 User Survey: Responses by Years of Experience 190
3.19 User Survey: Responses by Occupation 190
3.20 User Survey: Q.4 What document [markup] system[s] do you use most often? 191
3.21 User Survey: Q.6 What types of document[s] have you had most experience with? 192
3.22 User Survey: Q.7 What editing software do you prefer to use for structured documents? 192
3.23 User Survey: Q.8 How do you create a new document of the right type? 193
3.24 User Survey: Q.9 How do you give the title, author, and other key information for a new document? 194
3.25 User Survey: Q.10 How do you tell your editor to start a new section? 194
3.26 User Survey: Q.11 How do you apply formatting or styling? 195
3.27 User Survey: Q.12 How do you move blocks of text around when you edit a document? 196
3.28 User Survey: Q.13 How do you navigate around the document when editing? 197
3.29 User Survey: Q.14 How do you add blocks like tables, figures, lists, sidebars, etc? 197
3.30 User Survey: Q.15 How do you create or edit linking items? 198
3.31 User Survey: Q.16 How do you know what your document-in-progress will look like? 199
3.32 User Survey: Q.17 What best describes your general approach to creating and maintaining structured documents? 200
3.33 User Survey: Q.18 What specific feature(s) do you suggest others look for, or guard against? 201
3.34 User Survey: Q.19 What is the single most useful feature of the editing software that you use the most? 202
3.35 User Survey: Q.20 What is the single worst feature of any editing software for structured documents that you have used? 203
3.36 User Survey: Q.21 When using an editing system for structured documents, have you ever failed to find out how to do something that the product was actually capable of doing? 204

3.37 User Survey: Q.22 What features would you like to see in an editing system for structured documents that aren't in any of them at the moment? 205

4.1 Revised personas 221
4.2 Example of the base scenario screen 225
4.3 Effect of using Enter to break a line prematurely 231
4.4 Example modal dialog for adding vertical white-space 234
4.5 Manual alignment using multiple TAB characters and spaces 236
4.6 Modal dialog for adding horizontal white-space with the TAB key . . 238
4.7 Example modal dialog for adding horizontal white-space 239
4.8 Retention of styles after merging paragraphs 241
4.9 Ghosting of automatically-inserted elements in LyX 248
4.10 The 'standard' and 'extra' tool bars in the LyX editor, showing the 'semantic' approach to typographic variance (*A!* for emphasis and *A* with a human figure for personal names). There are another 20 or so toolbars for specialist purposes. 256
4.11 Adaptation of the I toolbar font button to drop down a menu 256
4.12 Experimental interface adaptation of the typeface menu to record styling requirements 257
4.13 Possible instantiation of a toolbar button for a labelled list 260
4.14 Navigation pane in an XML editor 263
4.15 Pasted text fragment with stub of overlapped markup 264
4.16 Some locations of a block cursor in a plain-text XML editor with markup revealed 267
4.17 Some locations of an I-bar cursor in a synchronous typographic XML editor with markup hidden 268
4.18 Conventional placement of footnote markup in DocBook and LaTeX . 270
4.19 Completion of a cross-reference in an XML system 273
4.20 Possible dialogs for adding a cross-reference 275
4.21 Responses to request for testers: annual number of documents written 294

5.1 Format of transcribed data (case 4) 298
5.2 Format of transcribed patterns (tasks 1 and 2) 299
5.3 Test data representation (case 4) 300
5.4 Pattern data representation (Task 2) 300
5.5 Test and pattern data arranged for matching 301
5.6 Number of testers and number of divergences from the expected patterns 303
5.7 Number of testers by task showing pattern eventually used 304
5.8 Number of testers by task showing affordance class 306
5.9 Completion of Test 1: new journal article 308
5.10 Completion of Test 2: add new paragraph 309
5.11 Completion of Test 3: split a paragraph 311
5.12 Completion of Test 4: join paragraph to preceding 312
5.13 Completion of Test 5: join paragraph to following 314

5.14 Completion of Test 6: add new section 316
5.15 Completion of Test 7: add new list 318
5.16 Completion of Test 8: move text block 320
5.17 Completion of Test 9: highlight product name 322
5.18 Completion of Test 10: add cross-reference 324
5.19 Completion of Test 11: add citation 326
5.20 Completion of Task 12: add document fragment 327
5.21 Total number of occasions of divergence by test 329

6.1 Arbitrary highlighted text 346
6.2 Deciding how to handle markup when deleting 348
6.3 Adding an item to the toolbar in the LaTeX editor *Kile* 351

List of Tables

3.1 Editors mentioned in the Expert survey 115
3.2 Expert survey: Q.2 Personal choice of editor for writing and editing (detail) 117
3.3 Expert survey: personal choice of editor for writing (Q.2b) according to software selection criteria (Q.2a) 118
3.4 Expert survey: personal choice of editor for editing (Q.2b) according to software selection criteria (Q.2a) 120
3.5 Expert survey: Useful features of an editor (Q.4) according to software selection criteria (Q.2a) 124
3.6 Expert survey: Poor features of an editor (Q.5) according to software selection criteria (Q.2a) 126
3.7 Expert survey: Features lacking in an editor (Q.6) according to software selection criteria (Q.2a) 128
3.8 Expert survey: Failure to identify abilities in an editor (Q.7) according to software selection criteria (Q.2a) 130
3.9 Software analysis: products examined 136
3.10 Software analysis (2005): identification of functions by product . . . 140
3.11 Software analysis (2009): identification of functions and their click/keystroke distance by product 150
3.12 Words and phrases used in retrieving messages 165
3.13 Classification of messages requesting editing software for structured documents (1990–2006) 169
3.14 Number of mentions of specific editors in messages requesting advice 171
3.15 User Survey: Estimate of population size exposed to the announcement of this survey, based on the extraction rate of messages in the Requests Analysis 186

4.1 Functions ranked against personas for perceived applicability to requirements 220
4.2 Responses to request for testers: Number of candidates by occupation and discipline 292
4.3 Responses to request for testers: Number of candidates by occupation and software used 293
4.4 Responses to request for testers: Number of candidates by occupation and types of document 293

5.1 Individual divergences (in italics) 330
5.2 Comments from testers 333

6.1 Initial formatting assumptions deducible from a DTD or Schema . . 359

I, Peter Flynn, certify that this thesis is my own work and has not been submitted for another degree at University College Cork or elsewhere.

Peter Flynn

Abstract

This research investigates some of the reasons for the reported difficulties experienced by writers when using editing software designed for structured documents.

The overall objective was to determine if there are aspects of the software interfaces which militate against optimal document construction by writers who are not computer experts, and to suggest possible remedies.

Studies were undertaken to explore the nature and extent of the difficulties, and to identify which components of the software interfaces are involved. A model of a revised user interface was tested, and some possible adaptations to the interface are proposed which may help overcome the difficulties.

The methodology comprised:

1. identification and description of the nature of a 'structured document' and what distinguishes it from other types of document used on computers;
2. isolation of the requirements of users of such documents, and the construction a set of personas which describe them;
3. evaluation of other work on the interaction between humans and computers, specifically in software for creating and editing structured documents;
4. estimation of the levels of adoption of the available software for editing structured documents and the reactions of existing users to it, with specific reference to difficulties encountered in using it;
5. examination of the software and identification of any mismatches between the expectations of users and the facilities provided by the software;
6. assessment of any physical or psychological factors in the reported difficulties experienced, and to determine what (if any) changes to the software might affect these.

The conclusions are that seven of the twelve modifications tested could contribute to an improvement in usability, effectiveness, and efficiency when writing structured text (new document selection; adding new sections and new lists; identifying key information typographically; the creation of cross-references and bibliographic references; and the inclusion of parts of other documents).

The remaining five were seen as more applicable to editing existing material than authoring new text (adding new elements; splitting and joining elements [before and after]; and moving block text).

To Teresina, Rachael, Thomas, and Olivia, for their patience and support.

Acknowledgements

A large number of people have contributed wittingly and unwittingly to this research, whose roots go back to a short paper I gave in the early 1990s drawing attention to the user interfaces of editors for SGML and LaTeX (Flynn, 1993).

Many friends and colleagues in the fields of usability testing and human-computer interaction, as well as in the XML and TeX fields gave their time and energy to the project with advice, warnings, code, and references. Their help has been invaluable and I thank them. In particular I thank the members of the principal usability mailing list, the participants of the Usenet newsgroups, and the various XML and TeX mailing lists whose assistance — particularly at short notice — was much appreciated.

Special thanks go to Professor Max Taylor, now retired, for taking me on in the first place, and to Professor John Groeger for his subsequent encouragement; to my supervisor, Dr Jurek Kirakowski of the Human Factors Research Group (HFRG), UCC; and to my colleagues in the Department of Applied Psychology and the Computer Centre (now IT Services) for their comments and encouragement.

I am also grateful to Google Corporation and Andy Arnt of their technical support team, who arranged for me to have extended access to their Usenet News database in order to undertake repeated searches which would otherwise have been blocked.

The unwitting contributors are the many now untraceable or anonymous posters to the related mailing lists and newsgroups over the last two decades whose questions, complaints, comments, and solutions formed part of the original instigation for this work. As it is now impossible to identify them individually, I thank them here *en masse*.

A select number have contributed both willingly and unwillingly: among the willing I would like to thank (in no particular order) Eve Maler, Bob DuCharme, Norman Walsh, Elaine Brennan, Debbie LaPeyre, Tommie Usdin, Julianne Nyhan, Professor Donnchadh Ó Corráin, Mavis Cournane, Lou Burnard, Michael Sperberg-McQueen, Barbara Beeton, Anna O'Connell, Mary Axford, Ang Gilham, Steve DeRose, Cris Bowers, Betty Harvey, John Chelsom, Karl Berry, Lauren Wood, Lance Carnes, Peter Flom, Lucy Lyons, and Sebastian Rahtz, all of whom have at various times gone out of their way to find information in their own fields which related to some aspect of this work.

I would like to thank all the authors who have been able to make their writings publicly available, despite the best efforts of the journal-publishing industry; and to the nascent efforts of Google and others to preserve for research the material that some publishers would otherwise have ignored.

I am particularly grateful to Barbara Beeton, who proofread the entire text in her inimitably thorough style, picking up on things which had survived all other

readings. All remaining faults are entirely mine.

The initially unwilling include the numerous software developers who responded to my online and offline nagging about the state of their software with explanations, justifications, excuses, and comments (and the occasional agreement!). At the time I risked being unduly harsh in my judgments, but I am grateful to the individuals concerned that once they understood that I was investigating, not attacking, they willingly and graciously gave their assistance.

Lastly, and most importantly, I thank my wife, family, and parents for their constant patience and support, and friends and relations both close and distant for acting as unofficial guinea-pigs and sounding-boards, and playing the part of 'user' so effectively.

Extended abstract

SUMMARY — This research investigates why non-technical writers appear to have difficulties using software for editing structured documents, and the low adoption rate of such software in environments where structured documents are written and used. It examines the background of structured documents, the nature of the software, and the measurements that can be made to test usability. Four studies were carried out among experts in the field, existing users, potential users asking about software, and with the software itself. A representative set of functions was derived for which an alternative interface or behaviour could be devised. These new functions were tested in a paper prototype by writers unaware of the existence of structured editors. The results indicate that making the functions do what the users expected, and making them easier to recognise, could lead to improvements in writing efficiency, and to greater effectiveness in the construction of a structured document without the user needing to see or know the markup.

Document structure has been an essential component in formal writing for many centuries. Readers rely on document structure to provide information about the size and shape of the document, the physical location of material, and its logical place in the argument. Writers and editors use document structure to provide a framework for authoring and modifying the text, to keep track of where material has been placed, and to monitor the development of the work. Structural identity can be marked up in many different ways, from the entirely visual (typographical) to the purely logical (hidden or symbolic). There are many software products implementing structural markup systems, but we concentrate on two of the most common, XML and LaTeX.

Software for writing and editing structured documents has been available in one form or another since the late 1960s, using markup (symbols) embedded in the text which both identified the structural function and triggered the appropriate formatting. Developments in the methods and logical rigour of markup during the 1970s and 1980s led to editing software which could provide guidance or prescription for the writer. This allowed organisational or editorial staff to control what structures could be used, and whereabouts they could occur, in order to provide consistency and improve the extent to which documents could be handled programmatically — in contrast to desktop publishing and wordprocessor

systems, which allowed the arbitrary styling of text to be the sole carrier of structural information.

The testing of software to measure its usability grew out of the study of ergonomics applied to early computers. In parallel with the growing use of desktop computers, the study of software usability emerged as a separate discipline, concentrating on the cognitive aspects instead of the physical ones. The modern (participatory, user-centred) model of software design has become a formal part of international standards, where it emphasises the use of measurement and the need for the design process to undergo consultative iterations, rather than being developed in isolation. In applying these precepts to interfaces to structured document editing, two key features are the degree to which users' experiences and beliefs condition their perception; and the effectiveness of affordances, or the extent to which what designers or software authors intend to be affordances really are affordances.

To provide background information, 20 experts were administered a questionnaire on editing software selection, installation, customisation, and training. The participants were chosen for their experience as consultants in delivering structured editing systems to industrial and institutional users. Their criteria for selecting software for clients emphasised stability, user-familiarity, and the ability to handle specific schemas — by contrast, the features of the software they felt most useful were configurability, a WYSIWYG view and a tree view, and the ability to run macros. Criticisms included poor styling, instability, lack of reviewing (change tracking), and lack of schema and catalog support. They tended to recommend software with an interface familiar to users, as this reduced training time, whereas their own preferences for software were different. They held a very critical view of editing software, viewing even the most recommended products as the 'best of a bad bunch'.

This duality (user-centric selection and technical criteria) influenced the survey of software, which was undertaken to provide answers to two questions that had arisen during the expert survey: how well does it support structured editing, and how easy is it for the user to operate the functions for handling structured documents? A third question arose as a by-product of the investigation, what differentiates the software? Thirty products were analysed in total, divided between two surveys (2005, and a repeat with updated software in 2009). Each one was tested with a sample text, and all the functions specific to structured-document handling were identified and exercised. In the 2009 survey,

the number of keystrokes or mouseclicks required to operate each function was also recorded. The 2005 survey showed that most products had most expected functions, but they differed in how they were named or labelled, and in where they were placed in the menus: none of the editors was really suited to use by authors without XML or LaTeX expertise. In 2009, the depth of each function was also recorded as a measure of **reachability** (effort). The mode was taken for groups of functions to provide comparability between products in whole keystrokes or clicks. Two XML and four LaTeX editors achieved a modal value of one (most functions one click away); most averaged two clicks; one scored three.

To determine the kind of editing software that users and intending users were looking for, an analysis was made of posts to the principal discussion forums (newsgroups and mailing lists). Messages were retrieved from logs and archives, matching against a set of keywords related to asking about available XML and LaTeX editors. An algorithm was developed to follow threads and comments back to the original message, and just over 400 of these original messages were then analysed to identify what the poster was asking for. Leaving aside the requests which were unrelated to usability, the most common specific requests were for WYSIWYG display, structure-view, ease of use, formatted-view, Unicode/language support, and strict DTD/Schema editing — and this despite leading manufacturers' claims that their products *are* easy to use, WYSIWYG, and typographically formatted. Despite the original emphasis on editors for structured documents, the three most-requested non-markup facilities were WYSIWYG, spell-checking, and change-tracking; and the last of these is commonplace in wordprocessors but technically challenging in XML or LaTeX.

The final study looked at existing users of structured-document editors, to see what software had actually been chosen, and how it was being used. Despite some misgivings over the stability and reliability of self-selected Internet questionnaires, a sample of over 60 professionals in their own fields was obtained. Markup experts, computing scientists, and technical authors were excluded as being too likely to possess in-depth knowledge of the domain which would skew the results away from the non-expert user base which forms our population. The background questions showed the sample to represent largely people with over 15 years experience in their field, about 70% in research and academia and 30% in business. The questions covered specific methods of working: metadata; sectioning; formatting; moving; navigating; adding tables, figures, and lists; linking; and previewing. It also asked about the features they found most and least useful, and what they would warn new users to look for. The results were

grouped according to the features and functions they related to, and compared with the data from the software survey and user requests study to create a ranked set of 14 operational requirements that could be tested in an interface

Two of the requirements (validity and Unicode) were excluded as being out of scope (they are in any event simple yes/no conditions). It was clear that the model interface would need to be a conventional-looking WYSIWYG screen, with the familiar toolbar and menus, and would work in a system that created a structured document. The many individual operations in the remaining 12 test areas were examined in detail, and a synthesis arrived at which allowed the key feature of each of the 12 items to be tested, using new or revised affordances (buttons, menus, icons). As writing an entire editor was outside the scope of the project, testing was done using a paper prototype, where printouts of screenshots were successively presented to the testers as they pointed to the affordance they would have clicked on. The tests were designed as a starting screen in a particular state, with a task for the tester to complete. Their choices were recorded on paper to provide a sequence of keystrokes or mouseclicks for each test. Twenty-one testers were recruited from within the author's institution, with backgrounds in engineering, computing, humanities, medicine, social sciences, and natural sciences. The tests were conducted in the usability laboratory of the Human Factors Research Group in the School of Applied Psychology at University College Cork in June 2013.

The sequences of keystrokes and mouseclicks for each tester for each test were compared with a set of sequences constructed beforehand representing possible solution patterns to each test. These represented various environments from classical text editors and wordprocessors to complex structured-document editors (existing affordances), and, critically, the canonical pattern using the new affordance[s] we provided in the prototype. All testers completed all the tests, although a small number of testers diverged from the the expected patterns in a few tests, and many of the recorded solutions used a mixture of existing and new affordances. The results of the comparisons showed that seven of the twelve modifications tested were completed using the new affordances by a majority of the testers (new document selection; adding new sections and new lists; identifying key information typographically; the creation of cross-references and bibliographic references; and the inclusion of parts of other documents). The other five were completed using largely existing affordances: these are seen as tasks more applicable to editing existing material than authoring new text (adding new elements; splitting and joining elements [before and after]; and

moving block text), and thus occurring less frequently during writing, and therefore not seen as critical. Overall, we conclude that modifications to the interface to present affordances that relate directly to the users' needs (rather than to the needs of the technology) would contribute to an improvement in usability, and thus to effectiveness and efficiency when writing structured text.

Terminology

There are several key phrases and words used here which have different meanings or emphases according to the domain or discipline. In order to avoid confusion I have therefore adhered to the prevailing conventions taken from Bradner (1997) which assign a strict interpretation to the key words 'must', 'must not', 'required', 'shall', 'shall not', 'should', 'should not', 'recommended', 'may', and 'optional', when EMPHASISED IN THIS MANNER. When shown in normal type, they retain their conventional contextual meanings.

Words in US spellings such as 'program' and 'dialog', are used to refer to a computer program, and the formal exchange of information between user and program, respectively, Their counterparts in British English spelled 'programme' and 'dialogue' retain their non-computing meanings.

Terms from the field of structured text and document engineering, especially SGML, XML, and LaTeX, are covered in the following list. These are working definitions for the purpose of this report, not the formal definitions used in the relevant standards, which are terse because of their reliance on a much larger specialist vocabulary which would be excessive to add here. The location of the formal definitions is given in [square brackets] where relevant.

Glossary

attribute (XML): an ancillary item of information or metadata embedded in the start-tag of an XML **element** for the purposes of identification, description, or effectivity, composed of a name and a value in quotes (for example, `status="draft"`)

character data (XML): text containing no [more] markup, either because there never was any (**character data content**), or because the **parser** has identified it all and passed that information on to the processor (**parsed character data** or `PCDATA`)

character data content (XML): the textual content of an element or an attribute that has no markup

character entity (XML): an entity defined to represent an individual character, typically one which cannot easily be typed on the user's keyboard, declared in a DTD for use in a document

character entity reference (XML): an instance of a character entity used in a document (for example, `↩` [↩], 'Left Arrow Hook')

command (LaTeX): a named identifier or instruction in procedural markup that causes some action to occur; in TeX, commonly used as a (slightly inaccurate) shorthand for **control sequence**, the formal name for such an

identifier, usually with an argument (text) in curly braces to output or act upon (for example, `\title{Glossary}`)

content model (XML): the formal statement in a DTD or schema specifying which element types are permitted or required to occur within the content of other elements, and (in **element content**) in what order and number (for example, a section containing a title and one or more paragraphs might be defined with the content model `(title,para+)`)

CSS (HTML/XML): Cascading Style Sheets is a W3C Recommendation for the method of applying typographic and layout styles to HTML and other XML-based structured documents.

document entity (XML): a portion of a document stored and referenced by name, usually a separate file

element (XML): an instance of an **element type** in actual use (that is, occurring in the body of a document), delimited by a start-tag and an end-tag, and containing text or other elements or both or neither. An element type which is explicitly defined to contain nothing (to act as a marker like HTML's `br`) is referred to as `EMPTY`, and is usually used in the special form of start-tag (`<br/>`).

element content (XML): a sequence or nesting of elements in an XML document at locations where no text is permitted, only other elements (in the **hierarchy** or **pool**; for example between paragraphs and the other components of a structural division, and between such divisions)

element type (XML): a named component of an XML document component, defined in a DTD or schema (for example, `section`; `para`). The suffix 'type' here means that we are talking about the element as a design pattern ('an element of type `para`'), not an *instance* of the element type in actual use in a document.

empty element (XML): an element which happens to have no content, neither text nor subelements, although it is permitted to (for example `<title></title>`)

EMPTY element (XML): an element which by design is not permitted to have any content, neither text nor subelements; identified with a slash before the closing angle-bracket (for example, `<pagebreak/>`)

end-tag (XML): the tag which ends an element: it is distinguished from a start-tag by a slash after the opening angle-bracket (for example, `</title>`)

environment (LaTeX): a named unit of document structure or description in LaTeX, containing text or other markup, and delimited by `\begin{...}` and `\end{...}` commands; approximately equivalent to an XML element in element content. The term was taken from *Scribe* (Reid, 1980a)

flow (XML): the selection of element types in an XML document that may contain character data (text). Typically they only occur within pool elements, where

the content model is said to be mixed (text and other elements)

Generic markup: Markup which describes portions of a text in terms of their general function or identity ('heading', 'chapter', 'paragraph', 'list', etc) instead of their formatting.

ghosting: the behaviour of an editor when creating a new element, displaying a mnenomic or prompt to remind the user what type of information is required. The ghost disappears the moment the user starts to enter text in the element. The technique is commonly seen in search boxes on web pages where the word 'Search' disappears when the cursor enters the box or the user starts to type

hierarchy (XML): the selection of element types forming the outermost structure in an XML document, characterised by forming a nested hierarchy of enclosures (for example, chapters containing sections containing subsections containing subsubsections). Hierarchy elements typically contain pool elements, but do not directly contain text

HTML: HyperText Markup Language, the markup used in web pages (and much else). Originally conceived as a small SGML tagset by Tim Berners-Lee at the Centre Européenne pour la Récherche Nucléaire (CERN), it was codified by the Internet Engineering Task Force (IETF) in 1993, and later evolved into XHTML and HTML5. Its widespread use (and abuse) was responsible for the rapid development of the World Wide Web.

HTML5: HyperText Markup Language v5 is a W3C Recommendation for future web development. Unlike its forbears (SGML, HTML, and XHTML), it does not have a formal definition as a Schema or DTD. It introduces extra elements for interactivity and presentation, and has also been adopted as the basis for the EPUB3 ebook standard.

markup: the tags, entity references, control sequences, escapes, commands, and other special values in a document which are not part of the text but which serve to identify the component parts of it, or specify what to do with it

mixed content (XML): a combination of character data and subelements which make up the content of conventional running text

null end-tag (XML): a null end-tag (NET) is the special form of abbreviated end-tag which may appear at the end of a start-tag when the element type is declared as EMPTY (for example the slash '/' in `<br/>`). Such forms are known as NET-enabled start-tags

pool (XML): the selection of element types available in a section or other structural division in an XML document, sometimes referred to as 'block' elements because their typical formatting employs a line-break before and after. Pool elements often contain text, but no text is permitted between them unless it is inside another element

processing instruction (XML): a special form of tag, not part of the document

structure, containing a value to be passed through unchanged to a subsequent process (for example, `<?css color:red?>`)

SGML: Standard Generalized Markup Language, the ISO standard (8879) for document markup upon which XML is based. Evolved from GML, codified by Charles Goldfarb and others at IBM in the late 1960s.

Smart Insertion: SI is a technique to overcome the reluctance or inability of a structured editor to insert a new element when requested, when the current cursor location indicates it would not be valid at that point. In outline, the behaviour is to move to the next location where such an element would be valid, and insert it there, creating such additional framing structure as needed: more details are in section 6.3.2 on page 344.

start-tag (XML): the **tag** which begins an **element** (for example, `<title>`); it may also contain **attributes**

subelement (XML): an element within the content of another element, whether in **element content** or **mixed content**

tag (XML): one component of an **element**: either a **start-tag** or an **end-tag**

tag abuse: the use of an element or command purely to get its formatted representation, its approximation to the actual need, or its availability (not being used anywhere else for its original purpose), because no adequate element or command is defined for the task

Target Markup Adoption: This is essentially a technique for purging text on the clipboard of irrelevant markup before pasting it, to avoid pasted text appearing with styles that are incorrect in context. The method is sometimes implemented in a non-robust manner as Paste Special in non-XML/LaTeX software such as web-embeddable HTML editors, but sometimes requires a modal dialog. The method is examined in section 6.3.1 on page 342.

XHTML: Extensible HyperText Markup Language is the XML version of HTML (which was, and — in its original form — remains, an SGML application). It was developed to return some rigour to web computing, and enable the machine-processing of web page content, which had been broken by the abuse of HTML.

XML: Extensible Markup Language, a proper subset of SGML originally designed to make it simpler to use SGML on the Web

CHAPTER 1

Background, scope, and methodology

1. BACKGROUND — Evidence for structure in writing — The use of structure — The document model — Markup theory and practice — Representation — Tag abuse — Definitions. 2. SCOPE OF THE RESEARCH — The nature of the problem — Constraints. 3. METHODOLOGIES AND TOOLS — Methodological development — Hybrid methodology — Persona analysis — Methodology and selection — Testing.

> *We have found a strange foot-print on the shores of the unknown. We have devised profound theories, one after another, to account for its origins. At last, we have succeeded in reconstructing the creature that made the foot-print. And Lo! it is our own.* (Eddington, 1920, p. 201)

This research investigates some aspects of the usability of editing software for structured documents, and suggests some possible changes to the interfaces to improve acceptance by writers and editors.

In section 1.1 we examine the background to the tasks involved in dealing with structured documents. We also identify the technologies covered by the types of software used. While much of this would be familiar to experts in the field of markup and document engineering, we have identified that the assumption of this foreknowledge on the part of the *user* (who has little formal knowledge of markup or document structures) is one of the principal inhibitors to the more widespread adoption of the technologies.

Section 1.2 explains the scope of the investigation, the populations that we considered, and the software examined. Not everything that writers write or editors edit falls within the definition of a 'structured document' (see section 1.1.7 on page 39). While the quantity of structured documents written on a computer could possibly be measured or estimated (we have not attempted to do so), the quantity of documents with no formal structure, or not requiring one, is probably inestimable. Similarly, the number of writers and editors engaged in structured writing is large but finite (again, unestimated); whereas every literate person with access to a computer is potentially a writer or editor of documents of some description. Lastly, while it is possible to write text documents with a very large range of programs, it is only practicable with a small number, and an even smaller number when the scope is restricted to those designed for structured documents.

The methodology we have adopted is discussed in section 1.3. The present investigation is closely related to other areas of HCI in which there has been extensive research into user interfaces (UIs), both in general and in specific classes of applications. However, with a few exceptions which we examine in Chapter 2, there appear to be no specific methodological adaptations applicable to research on the interfaces for structured documents.

Chapter 2 examines the published work on the psychology of editing structured documents, with specific emphasis on research into usability. There is an extensive literature on the usability of computer interfaces, which substantially informs the overall interaction design of modern programs, including those dealing with structured documents. By contrast, there is rather less relating specifically to the demands of structured document editing, although it has been investigated in depth by a few authors. Against this must be set the extensive body of research into general text editing, and into the application to text editing of analytical methodologies developed in the early 1980s such as Goals, Operators, Methods, and Selection Rules (GOMS). Some research from other disciplines is also examined, particularly from computing science, Humanities computing, and the field of publishing, all of which have close points of contact with the immediate topic.

In Chapter 3 we investigate the users' perceptions of the tasks, and the facilities offered by editing software. Four enquiries were made:

1. a survey of expert practitioners in the field;
2. an analysis of the software and its features;
3. an analysis of requests for advice posted to specialist discussion forums;

4. a survey of authors and editors of structured documents.

Perceptions were measured both quantitatively and qualitatively, based on expectation (as expressed in the requests for help) and on behaviour (as expressed in the final survey).

Chapter 4 compares these expectations and behaviours with the facilities offered by the editing software analysed in the second enquiry above. This leads to the construction of a model of the differences, from which we derive a set of changes to the interface to address the disparities between expectation, behaviour, and software performance. A paper prototype of these changes was tested where it was possible to implement them within the abilities of known software, and the results of testing are reported.

In Chapter 5 we present the results of the tests, and in Chapter 6 we draw some conclusions, and make suggestions for further work.

1.1 Background: structured documents

The term 'structured documents' is widely used in the fields of markup and document engineering with the recognised sense of a document which exhibits a conscious division and subdivision of the material into a regular pattern of reoccurring units which form an ordered hierarchy (see section 1.1.7 on page 39). A structured document generally says nothing about the form in which it will appear, which may even not yet have been decided (Reid (1989) calls this a case of 'late binding').

Outside the fields of markup and document engineering it is arguable philosophically that all documents are structured in some way (physically, semantically, linguistically, syntactically, or according to some other pattern), but we restrict our study here to those documents requiring (or exhibiting evidence of) deliberate physical organisation in a form that can be used by others without the author's specialist knowledge of the subject matter.

For example, if the Abstract of a paper on biochemistry is clearly labelled as 'Abstract', then an editor, typesetter, librarian, funding agency, or anyone else can immediately identify it, extract it, copy it, print it, or add it to a database, without having to be a biochemist themselves; this would not be the case if the abstract were a paragraph indistinguishable from all the others in the paper.

There is evidence from the historical context which we present below that intellectual material judged by an author to be of significance to others was seen to deserve an explicit or implicit exposition of its structure. Labelling and classification are seen as essential to human existence: Kirakowski and Corbett (1990, p. 46) hold that we are 'essentially an order-invoking animal'; a view reflected in another context: "[O]ur delight in the order we find [. . .] reflects man's preoccupation with pattern" (Papanek, 1972, p. 4).

1.1.1 Evidence for structure in writing

The written word is generally considered to have evolved some 5,500 years ago in the Mesopotamian cultures. Extant records use incised or impressed letters on clay tablets, and the majority appear to be narrative sagas or business documents (Gaur, 1992).

Structure in documents is evident from an early stage. The Calendar of Gezer (Figure 1.1 on the facing page) is a clay tablet from 925BC with incised writing, showing a list of farming tasks for each month, with each item starting a new line, reading right to left; a form which a modern author would immediately recognise as an itemised list, (Wilson, 2007, pp. 283–284).

A similar layout is seen in Roman military reports (Figure 1.2 on page 6), where the items — an inventory of men at various tasks in the workshops — start new lines (*structores. . .*, *ad plumbum. . .*, *ad furnaces*, etc) (Centre for the Study of Ancient Documents, 2003).

There have nevertheless been periods when structure as we would recognise it today was not obvious to the untrained eye. Scribal cues are either absent, or are contextual, and thus only comprehensible if you can read the document and have been trained to decipher them. In the manuscript era, the cost of vellum dictated that abbreviations and pothooks be used to save space; some of these became codified into a type of markup — the ampersand (&) as an abbreviation for Latin 'et' is not only a linguistic conjunction but also a visible marker between two pieces of text.

In the example of a legal document in Figure 1.3 on page 7, the clauses are marked by item, but the myriad references and commentaries are placed in the margins, partly by tradition and partly to keep them close to the text; their identity and point of attachment cannot easily be resolved by the non-expert.

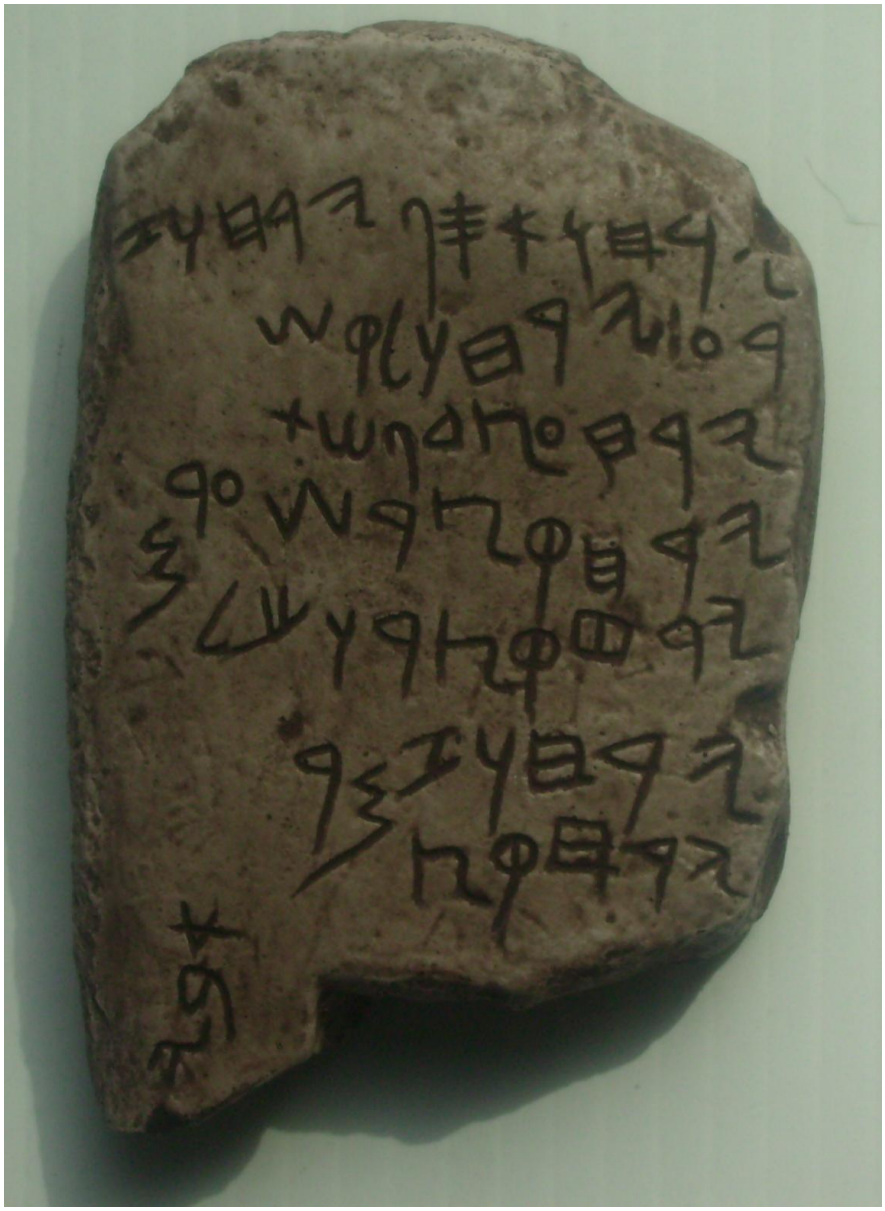


Image courtesy of the Herries Collection © 2007 Herries Press; reproduced by kind permission of Dr Peter Wilson.

A transcription (with translation) shows the list to concern the timings of the seasons and harvests. (Wilson, 2007)

Figure 1.1: Clay tablet calendar, Gezer, *c*.925BC

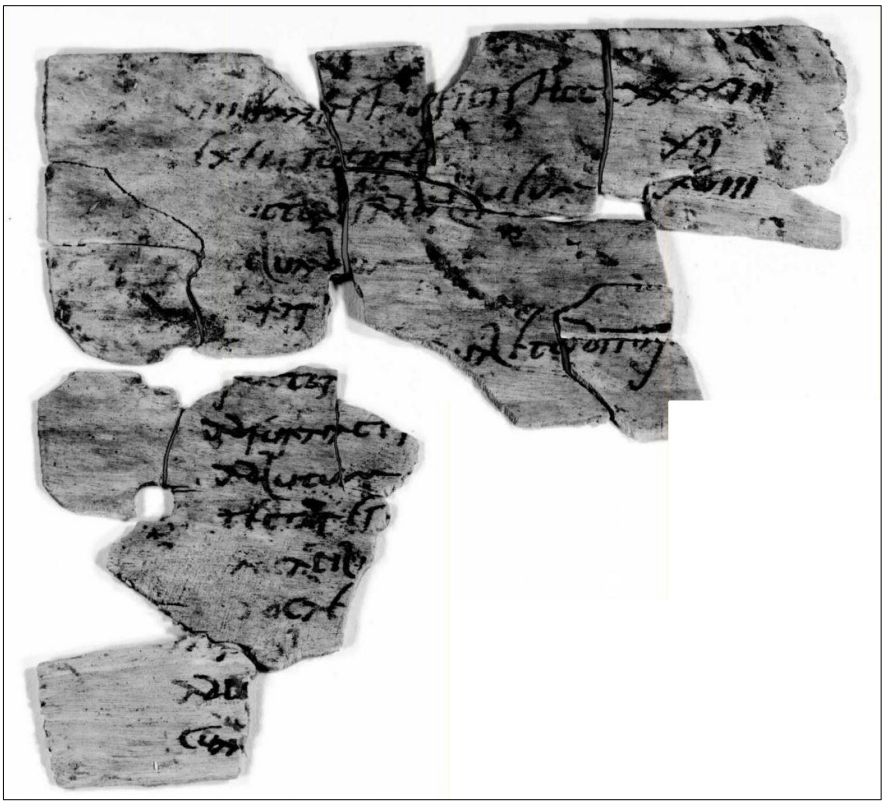


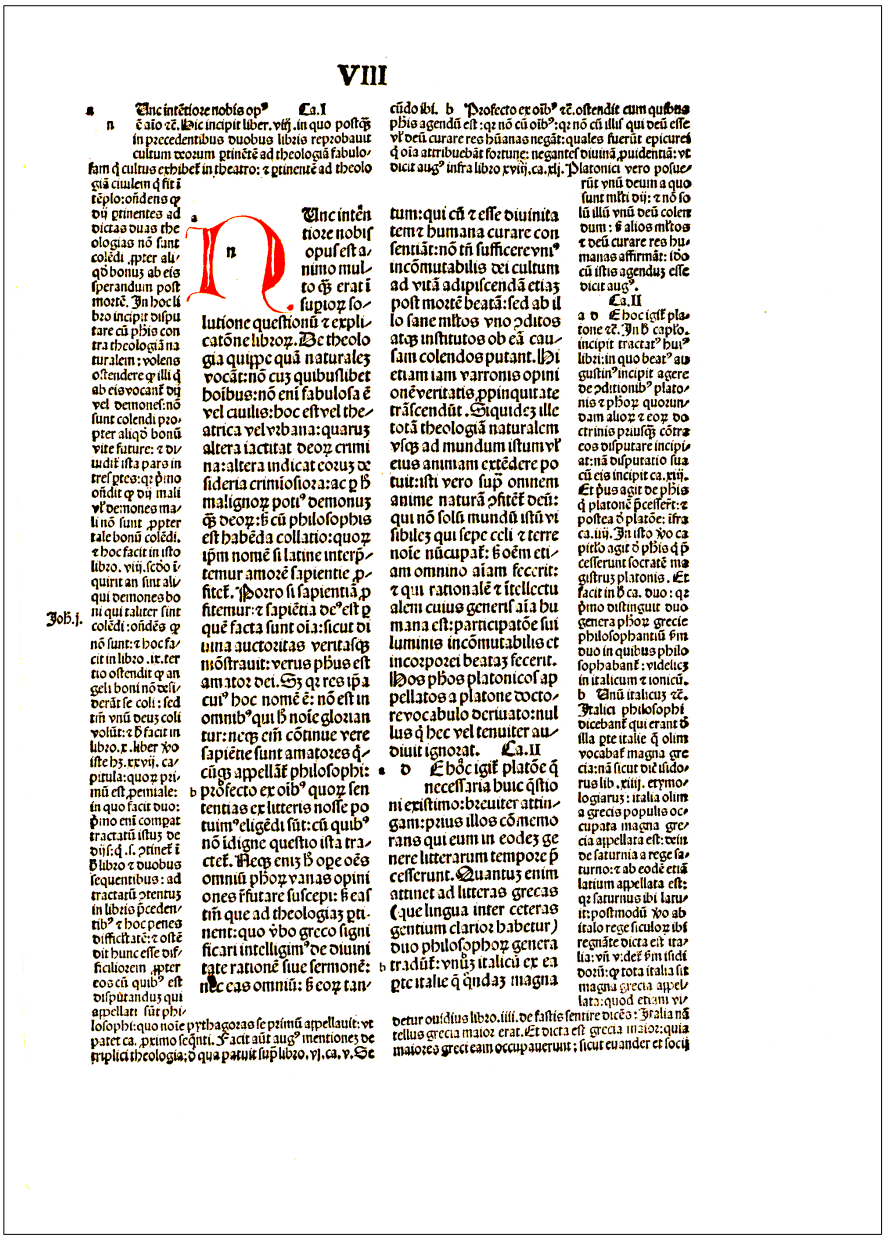
Image copyright of the Centre for the Study of Ancient Documents, The British Museum and other copyright holders.

The Vindolanda writing tablets were excavated from the Roman fort at Vindolanda near Bardon Mill on Hadrian's Wall from 1970 onwards.

Figure 1.2: Roman military report from Vindolanda (tablet 155)

Legal documents, both canon and civil, as well as biblical documents, had long used their own separate sets of conventions, particularly in the use of colour to distinguish major and minor divisions. The notion that the *format* of a document should educate or elucidate the text gradually became accepted, with leading thinkers and writers like Oresme redesigning the conventional document layouts in his translations of Aristotle in order to help the reader [according to Kirschenbaum (2005, p. 153), citing Sherman (1995)].

With the establishment of printing as the means of mass communication, editions of complex scientific texts were using structures clearly recognisable to us. Figure 1.4 on page 9 shows a page of the *Fabrica* of Vesalius, with a chapter title and number, a figure cross-referenced to the chapter number, containing a table of call-outs explaining the images, with further cross-references to other parts of the document. While Vesalius was a notorious perfectionist, it is significant that less than a century after Gutenberg, he and his printer/publisher


VIII

Legal text of the 13th? century

This layout shows the placement of commentary and notes in the surrounding margins. The margins were regarded as an integral part of the structure, as they allowed the owner to add further comments in manuscript. The type designer Jan Tschichold analysed the connection between text, commentary, and margins of this style, which exhibit a use of the Golden Section.

(Tschichold, 1951, pp. 12–13)

Figure 1.3: Structure in early legal documents

found it perfectly acceptable to use such structures without explanation or apology to the reader, although he did take extreme pains to ensure that his printer got it right (Saunders & O'Malley, 1950, p. 46).

1.1.2 The use of structure

Until the formalisation of **generic markup** in the 1980s (ISO 8879:1985, 1985), most writers' use of structure was largely conditioned by its representation in **formatting** and **layout**, as these were — and remain — the conventional ways of communicating structure to the (human) reader.

Formatting: the typographic instantiation of text, using a choice of typeface, size, weight, style, spacing, and symbols

Layout: the pattern of visual arrangement on the page, which may include normal continuous text, illustrations, figures, lists, and other elements which go to make up the document

The author or editor would choose a particular combination of formatting and layout to carry specific information. In western conventions, a new page, with a number and title prominently formatted, signals 'a new chapter starts here'; a sequence of indented paragraphs each preceded by a consistent symbol indicates a list of items for action, summary, reference, or some other purpose; a superior digit after a word, partnered by another at the foot of the page, means that the text at foot is some relevant comment on the argument, perhaps a reference or an explanation.

While the semantics of these devices depends on culture and experience, the majority of them can safely be assumed to be recognisable by the educated reader in their author's field, *and to be unambiguous given their context*. The author or editor applies the relevant formatting to a word or phrase (or a block of text) with the keyboard or the mouse, and the wordprocessor makes the necessary changes to the appearance.[1]

Any document created this way contains an exclusively visual representation of the structure. It requires a human to interpret it because the meaning of the formatting and layout depends on the context and the relative position and sizes

[1] In earlier times, the typesetter followed the 'markup', which was the proofreader's marks or editor's marginal annotations describing corrections to the formatting and layout required at a given point.

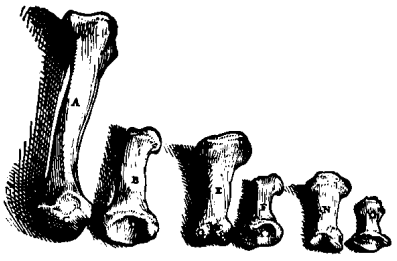
148 ANDR. VESALII DE CORPORIS

& parem inibi cum ossibus superficiem constituentes,illorum gibbum,quod depressum magis,ac planum,quàm teres cernitur,ne minimum quidem excedunt.adeò,ut quemadmodum ulnam & radium attensis ipsis cedere ostendimus musculis,sic quoq; posthbrachialis ossa secundùm lõgitudinem angulosa, & lineis quibusdam pulchrè extuberantia spectentur, pro musculorum uidelicet,qui ipsis exporriguntur,ratione.uti etiam posthbrachialis os,indicem sustinens, in internã manus sedem, quà brachiali articulatur, notatu dignum educit[k] processum,cui[l] musculus brachiale mouentium secundus inseritur.Dein posthbrachialis os,paruo præpositum digito,quà brachiali committitur,externo suo latere, ueluti à brachiali[m] protuberat, ut cõmodè tendinis insertionem admitteret[n] musculi brachiale mouentium tertij.Insuper ossa indicem & medium digitos suffulcientia,ipsorum externa sede ad brachiale[o] ampla quoq; & aspera sunt, ut bifidum excipiant tendinem[p] musculi brachiale mouentium quarti,qui ideo potissimum uidetur bipartiri, & dein latescere, quòd simplex & teres uni alicui processui parũ aptè insereretur, qui extra brachialis & posthbrachialis exteriorem gibbamq; sedem extuberasset.

k 1 fi.cap. 25 supra charact. 6 & 7.
l 8 musc. tab. A.
m 1,2 fig. cap. 21 supra N.
n 9 musc. tab. E.
o 2 fi.cap. 25 supra charact. 5, 6, 7.
p 11 musc. tab. A.

DE OSSIBVS DIGITORVM MANVS.
Caput XXVII.

FIGURA VIGESIMI SEPTIMI CAPITIS, TRES CONtinens tabellas.



PRIMA TABELLA. SECVNDA. TERTIA.

TRIVM TABELLARVM HVIVS CAPITIS figuræ,& earundem characterum Index.

Ossium digitorum seriem duæ priores figuræ, ad initium uigesimi quinti Capitis repositæ, commonstrant: quemadmodum & tres integrum ossium contextum exprimentes figuræ, ad huius libri calcem locandæ. Figura autem hic occurrens, ac in tres ueluti tabellas digesta, digitorum articulos oculis subjcit. Harum e-
A. nim prima, duo exprimit ossa: quorum alterum A insignitum, estq; posthbrachialis os, indicem sustinens. alterum
B. B indicatur, estq; primum indicis os ab externa sede, quemadmodum & illud posthbrachialis os, delineatum.
C. Præsenti itaq; tabella, primi digitorum articuli effigies proponitur. C enim caput rotundum notat posthbrachia-
D. lis ossis. D uerò sinum primi ossis, quo caput C insignitum excipitur.
E.F. Secunda tabella primum indicis quoq; os notat, E notatum, & secundum pariter F insignitum: ut horum
G,H. ossium delineatio secundi quatuor digitorum articuli speciem proponeret. G itaq; & H duo indicant capitula
I,K. primi indicis ossis. I uerò & K sinus secundi ossis, quos duo primi ossis capitula subintrant. L autem sinum si-
L. gnificat, in medio capitulorum primi ossis consistentem, ac in quem tuberculum secundi ossis inter duos sinus I
M. & K notatos prominens, & M insignitum subingreditur.
N.O. Tertia tabella secundum indicis os N insignitum, & tertium O notatum proponit, tertij cuiusq; digiti articuli

A page from Vesalius' *Humani Corporis Fabrica* (Basle, 1543)

The now-familiar modern structure of chapter, figure, table, marginal notes, and cross-references are clearly visible (Lawson, 1990, p. 131). Vesalius' letter to his printer, Oporinus, gives minute details of the corrections and placement necessary for the text, figures, and tables, which he calls the 'Index of the characters' (cross-reference letters), as well as specifications for the marginal notes. (Saunders & O'Malley, 1950, p. 46)

Figure 1.4: Structure in early scientific books

of the text, headings, lists, etc, as shown in Figure 1.5 on the facing page. Such a document contains no information that a computer can reliably use to determine where the chapters start, what the significance of a list is, or what a raised digit is there for. Large bold text at the top of a page could mean a dozen things; as could indented text, symbols or no; and a raised digit could be a note marker or a mathematical power.

The use of **generic markup** (which we summarise in section 1.1.4 on page 24) places formatting and layout at one remove from the author and editor. Instead of using the controls of the program to apply bold, italics, spacing, size changes, and other effects directly, the author or editor attaches a label to the relevant text which tells the computer what it is (a chapter title, a list item, a footnoted reference, a foreign word). The program responds by instantiating the correct formatting according to the stylesheet associated with the document type being used — there is, in effect, no difference in the visual result for the user. But in the process, the program has acquired an essential piece of information: the identity of that element of the document.

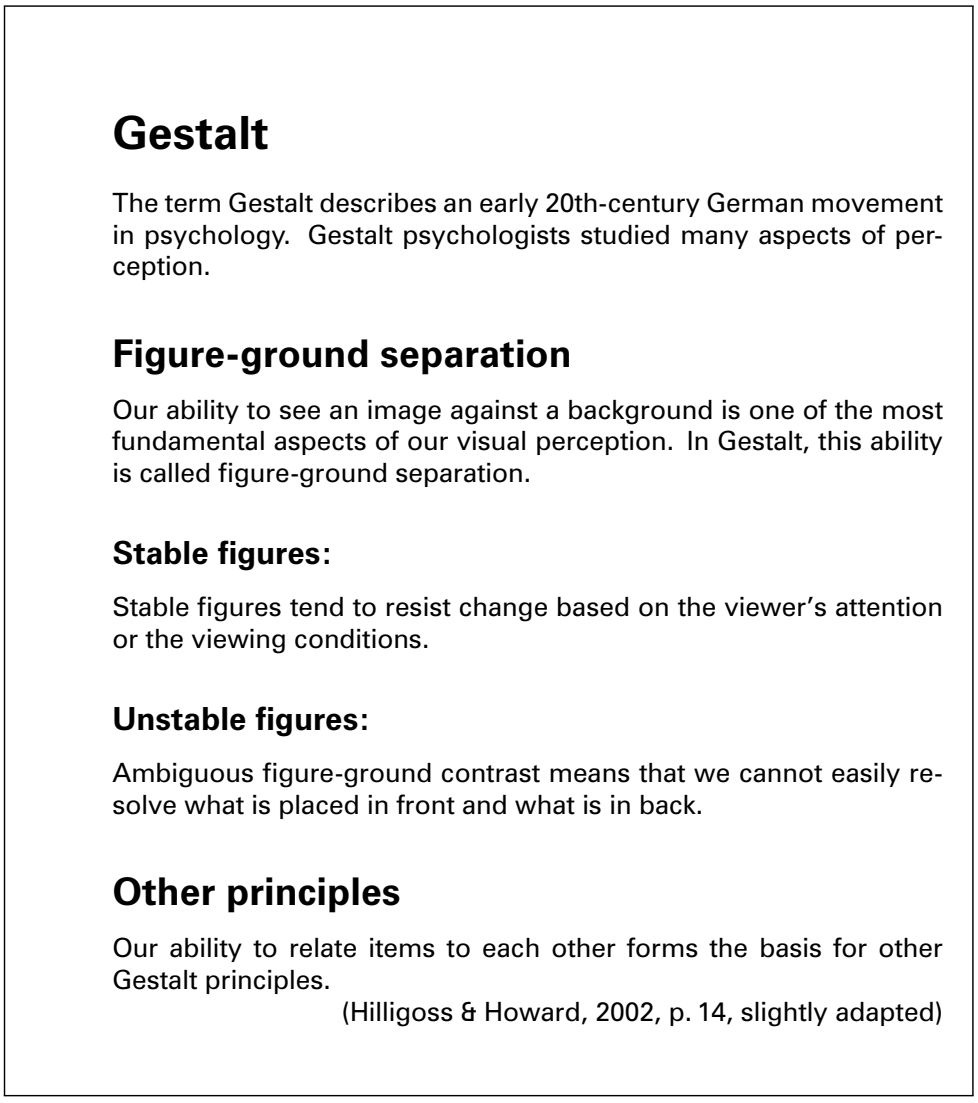
There are both benefits and drawbacks for the author or editor using this method. The benefits include:

- The formatting and layout are completely consistent throughout the document (provided, of course, that the author or editor has used the correct labels); consistency is a pre-eminent requirement of all publishers;
- If a particular format or layout needs changing (for example, to redesign some aspect of the document), all that is needed is a change to the specification of the style, and all instances of that style will immediately change appearance to conform;
- An entirely different style can be applied, matching the same labels to a new format or layout, in order to render the document in a different way for a new purpose, without the document itself having to be edited in any way — for web publication, for example, or for another journal or publisher, or for a specialist requirement such as voice or Braille;
- All forms of cross-referencing and indexing can be automated, so that when text is moved during editing, or changes are made to the formatting and layout, the points of reference will always resolve correctly; that is, footnotes will be on the right page; references to page or section numbers will adjust themselves; and chapters will still start on a right-hand page;

- any subsequent computer-based searching or browsing can return a valid reference to the place located within the document.

By contrast, there are several disadvantages, particularly when seen from the author's perspective:

- The author is not in charge of æsthetics: the use of a stylesheet precludes the author (or editor) from any design decision. While this is understood when writing for a specific publisher with a strict house style, it is unacceptable when writing for oneself or for an organisation with no such rules — and most editing software allows the writer to change the styling (but see 'Interventional view of structure' in the list below (page 17));



Gestalt

The term Gestalt describes an early 20th-century German movement in psychology. Gestalt psychologists studied many aspects of perception.

Figure-ground separation

Our ability to see an image against a background is one of the most fundamental aspects of our visual perception. In Gestalt, this ability is called figure-ground separation.

Stable figures:

Stable figures tend to resist change based on the viewer's attention or the viewing conditions.

Unstable figures:

Ambiguous figure-ground contrast means that we cannot easily resolve what is placed in front and what is in back.

Other principles

Our ability to relate items to each other forms the basis for other Gestalt principles.

(Hilligoss & Howard, 2002, p. 14, slightly adapted)

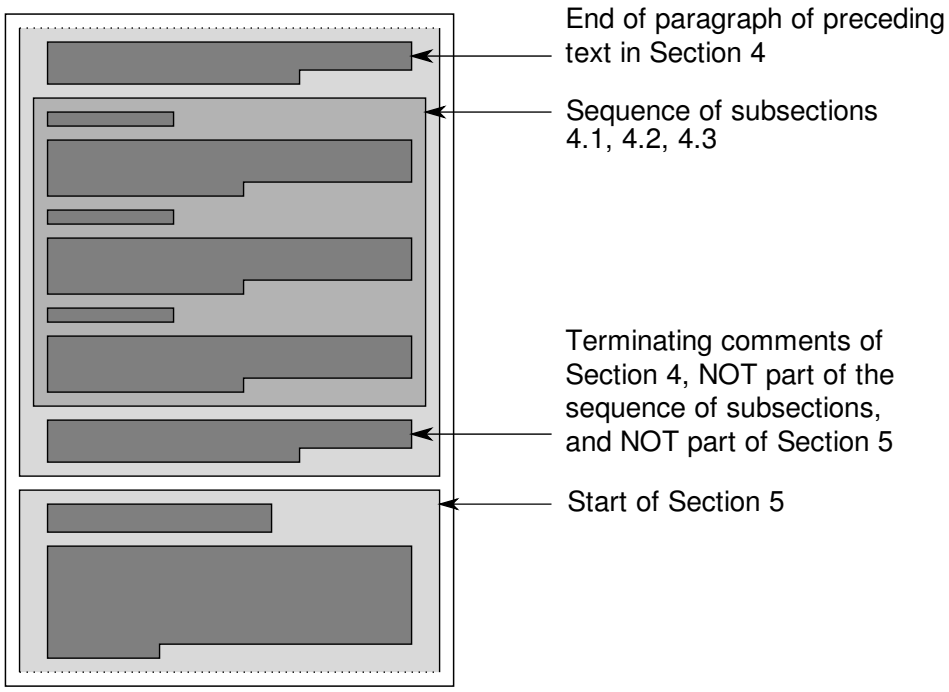
Figure 1.5: Text layers and hierarchy, showing the descending size and position of headings representing the sections

- It sometimes happens that there is no way for an author to format something correctly: it is not provided for by the stylesheet (that is, by selecting an appropriate markup for the task). If an author wants a certain passage to appear in a specific layout, and that markup/stylesheet combination has not been provided for, dissatisfaction or rejection may arise, leading to **tag abuse** (see below);
- Where there *is* a choice of formats (based on markup), the author may wish to choose one different from how the document type designer envisaged it to be used, resulting in the underlying markup misrepresenting the intention of either the author or the designer;
- There may be occasions when the author requires a specific structure which is unavailable in the document type being used. A common request is for paragraphs following a sequence of subsections, where such paragraphs are seen as part of the containing section, in the nature of *obiter dicta*, and not a separate, final subsection. This is illustrated diagrammatically in Figure 1.6 on the facing page;
- The repurposing described in the list of benefits above, which has long been a Holy Grail for both publishers and markup theorists, is largely unachievable within the limitations of current software and acceptable typographic standards (discussed further below).

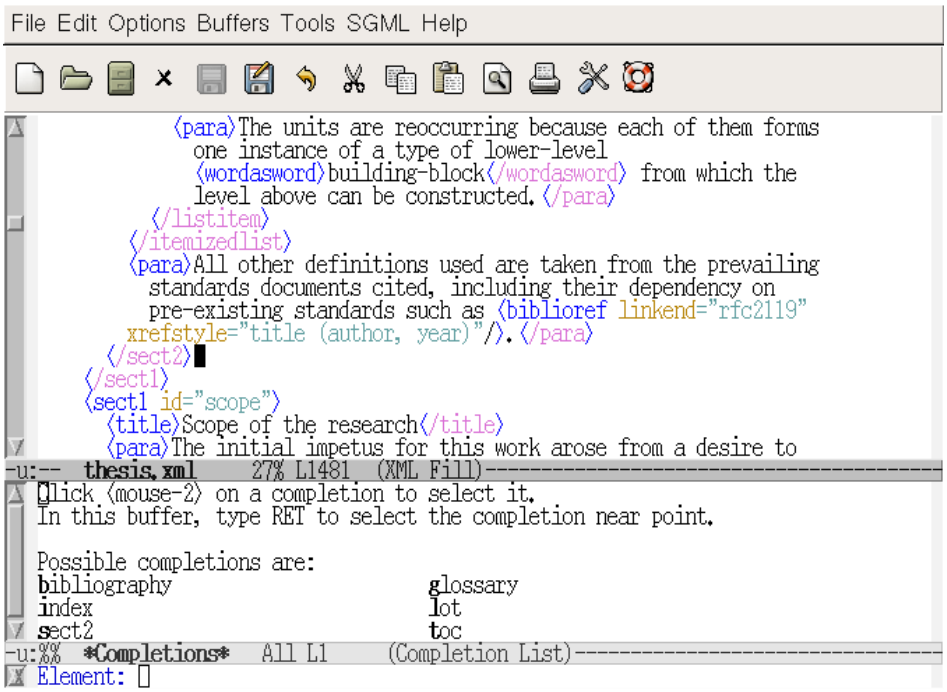
In many of these conflicts between structural/stylistic control and authorial freedom, the most common solution is for the author to use whatever markup or styling is actually available, regardless of whether or not it was designed for the current purpose. This is a phenomenon known as **tag abuse**, and is discussed in more detail in section 1.1.6 on page 37. A common example, familiar to anyone editing a website in HTML or a document in LaTeX, is the conflation of markup for *emphasis* with that for undistinguished[2] *italics*.

The formal solution usually recommended is for document type designers to study or monitor authors' evolving requirements more closely, and to provide markup for the purpose; or for documentation and training to explain precisely why

[2]This term from the vocabulary of formal data analysis has a slightly different meaning from the conventional usage of 'ordinary' or 'unremarkable': it implies that there is a parallel 'distinguished' version in existence (similar terms 'marked' and 'unmarked' are employed in linguistics, but are avoided in the context of markup). The present author attributes this usage to a perceived need to avoid the word 'distinctive', as it has no obvious negative; the X.500 Directory standard appears to be its first use with this meaning [distinguishedName] (Ellison et al., 1999).



a) This structure is not available in many document types: between the end of the final subsection and the end of its containing section, no further text is allowed, because there is no recognised way to distinguish it visually from the final paragraph of the final subsection which precedes it.



b) A validating XML editor (here, Emacs/psgml) using the DocBook DTD, showing the only allowable markup at this location (after the last level 2 subsection, before the end of the level 1 section); this does not include a paragraph.

Figure 1.6: Orphaned paragraph (a) floating between the end of a subsection and the end of a section, and (b) an attempt to add material in this position in an XML document, showing that it is not possible (in DocBook)

certain features are disallowed in some circumstances (Maler & el Andaloussi, 1999).

The application of classical markup theory to text (Coombs, Renear, & DeRose, 1987; Sperberg-McQueen, Huitfeld, & Renear, 2000; DeRose, Durand, Mylonas, & Renear, 1990) implies that these advantages should all be attainable without the need to change the markup (the labels in the document) in any way — leaving aside correction or updating, which are purposes in their own right — because the markup should always carry sufficient additional information (**metadata**) to enable subsequent re-use.

This immediately triggers a major objection: most documents are not adequately marked up, for many possible reasons: a lack of time or expertise; a lack of suitable software to apply re-usable markup effectively; a lack of coordination between members of a team; because the advantages are not fully appreciated; or unresolved conflicts between the way something is marked-up for one purpose, and its need to be done differently for another. In some classes of informal document, detailed markup is simply unnecessary, because the implicit shared cultural referents are well-established. In others, it is, to be frank, a mess: the ill-fated standard ISO 11179:1997: 'Specification and Standardization of Data Elements' was made available in separate chapters, each in a different file format (*WordPerfect*, SGML, *Word*, Postscript, PDF).

The sticky office *Post-It* note, for example, is available in plain yellow and can carry any message the writer cares to inscribe. In the 20 years it has been in use, office workers everywhere have become accustomed to recognise it stuck on their furniture as a sign that someone has written some information on it for them. In effect, it is its own markup, saying 'Message for you' or "Don't forget…". But it is also available in white, preprinted with 'Phone Message', and with fields for the caller's name, number, date, time, and topic. The first is markup-free and has universal applicability. The second is for a special purpose, and the markup has been designed to prompt the writer not to forget to include key information.

Given the seemingly unattainable nature of fully re-usable markup, considerable attention has been paid to the use of logic, heuristics, and statistics to automate the process (Kelly & Abrahamson, 1991; Taghva, Condit, & Borsack, 1995; Abolhassani, Fuhr, & Gövert, 2003). However, while systems have been developed for 'vertical' applications such as news article markup (Haake, Huser, & Reichenberger, 1994), journal article markup (ANSI/NISO Z39.96-2012, 2012), and many others, to date there is no general-purpose system available to

implement the techniques for an arbitrary range of documents, and this is perhaps an unattainable target.

It is nevertheless true that there is usually some degree of structure evident within even simple documents, such as a blank line or indentation to indicate a new paragraph, or a large font to indicate a heading, and it is possible to develop ad-hoc systems using simple toolsets such as the Unix text tools to impose rudimentary but sufficient markup to enable documents to be opened in an XML or LaTeX editor, and leave the finer detail to human editing. Interpreting in any greater detail the arbitrary and inconsistent nature of manually-applied formatting and layout, given its high level of context-dependency, remains a subject for further work in the field of Information Retrieval. As some of the authors in this area have noted:

> Further, all these heuristics become useless and the difficulties we have mentioned multiply, if the device fails to zone a page properly. For example, the title-finding module will not find the title if, in a two column document, its zoning order follows the first column of text. This may occur if a document's title is right justified; also, if a floating object is zoned improperly, any object recognised by *Autotag* may be identified incorrectly. Proper zoning is a prerequisite for correct *Autotag* output. (Taghva et al., 1995, p. 325)

(We identify in section 4.3.4.1 on page 257 a related problem with the use of Named Styles when dealing with wordprocessor document.)

While physical structural simplicity may be evident in some classes of documents (novels, for example), at the opposite extreme there are a number of applications of substantial complexity for markup, among them technical documentation, in which adherence to a predetermined structure is often compulsory and may be life-critical, such as aircraft or medical equipment maintenance. In legislative drafting, the structural depth of the subsectioning may extend to eight or more levels deep; and literary and linguistic analysis has at its disposal the entire bestiary of *apparatus criticus* and other epexegetic impedimenta.

This kind of structure can be indicated in markup in different ways, but when writing or editing with a computer, the facilities for applying these indications vary hugely in their accuracy and effect. We shall examine these in section 1.1.4 on page 24.

There is therefore a tension between how well the structure can be represented, and how easy it is to do it; and this tension is often unrecognised by writers, who

may have their own internal model of the document as well as their own internal model of what their editing software is supposed to do. As we shall see in section 3.1 on page 107, the resulting inconsistency appears to be a major cause of the difficulties experienced by editors and other subsequent users of a document. The balance between effectiveness and ease of use is at the core of this research.

1.1.3 The document model

We have seen that the structured document as we know it today has a clear ancestry, and — in the western world — an accepted and conventional pattern for implementation.

1.1.3.1 Structure

The technical, business, or academic writer typically uses the term 'structure' to describe the physical division of the text into the logically-ordered hierarchy of topics, subtopics, illustrations, and other features with which their audience is familiar, and which we codify in our definition of the structured document in section 1.1.7 on page 39.

However, the term can be interpreted differently by those who write, read, edit, typeset, interpret, or otherwise process such documents. Several other views of 'structure' exist in the formation and analysis of documents; the three most common of which we shall term linear, literary, and interventional.

Linear view of structure: The linear view treats the document as sequential narrative or a unidirectional flow, and its division into chapters, sections, and paragraphs as simply milestones or a recognition of convention (Plotnik, 1984, pp. 22–23). There is considerable validity in this view, especially insofar as the *reader* of narrative literature or drama is concerned, where the whole purpose is to start at the beginning and go on to the end; as distinct from a business, technical, pedagogical, or reference document, which has different requirements for cross-referencing and navigation.

Literary view of structure: The author of literary material considers the word 'structure' in terms of plot, revelation, character development, narrative pace, interaction, suspense, dialogue, and other devices of literary

construction. This is not a challenge to our view of structure, but simply a different meaning for the same word (see Chapter 2 on page 56). Nevertheless, it has relevance when examining the responses of some authors to enquiries about 'structure' and their use of software.

Interventional view of structure: The desire of authors to present their material in a certain way has been recognised as a potential conflict with the needs of the editor or designer to present the reader with a readable document; certainly since the days of metal type (Fyffe, 1969, p. 55). The document engineer also has an obligation to maintain a file which can be processed reliably into print, web pages, online PDFs, Braille, voice, and other formats.

The interventional view has specific relevance to the analysis of editing as a function. As we have identified in the list on page 11, the conflict may arise when the author believes that a certain portion or aspect of the text must be presented in a certain way, and that this is not provided for in the interface, and also happens to break the hierarchical 'tree' model (discussed below). In effect, 'the author sees it as *his own job* to invent the schema' (Piez & Usdin, 2007, my emphasis) — a close parallel with the stance taken by Vesalius which we saw in Figure 1.4 on page 9, and a tendency (unrelated to schemas) noted earlier in Owston, Murphy, and Wideman (1992, p. 271). Norrish (1989) describes it well from the point of view of production experts:

> Traditionally the design of documents was carried out after the document was written, by people highly trained and skilled in editing, designing and preparing manuscripts for production. Computer-based systems however give the originator the opportunity to take the document through all the stages needed for production. This causes problems, as originators are usually not trained in editing and design skills and have rarely had to produce a great variety of realizations of text structures, although they are able to use and understand them. The models of text which are offered to those originating and producing documents on computer-based systems are criticized by typographers for being over-simplistic and appearing to pay scant attention to the appearance of real documents and the needs of the user. The typographer's worry is that the clarity, consistency and diversity with which structures can be realized will be lost and the reader will suffer in consequence.

Intervention may occur on any side: the author, editor, designer, or typesetter may

try to insist on a change which would cause such a break.

These views of the meaning of structure are by no means mutually exclusive, and they will frequently be found to overlap. While they do not significantly impinge on the meaning of 'structured document' as we will use it, there are some cases where they imply requirements which may conflict with the use of editing software, and for this reason it is necessary to take them into account.

To clarify the conventional model of a structured text document, there is a set of units which we will use to describe the physical divisions of the document. These units come in three classes: **hierarchy**, **pool**, and **flow**.

Hierarchy: Titling, legal and copyright statements, tables of contents, prefaces, forewords, parts, chapters, sections, subsections, appendices, bibliographies, glossaries, indexes…

These may occur or reoccur in family-tree form, but *cannot intermingle* (you cannot have an appendix in a subsection, or a chapter in an index, or a preface in a glossary).

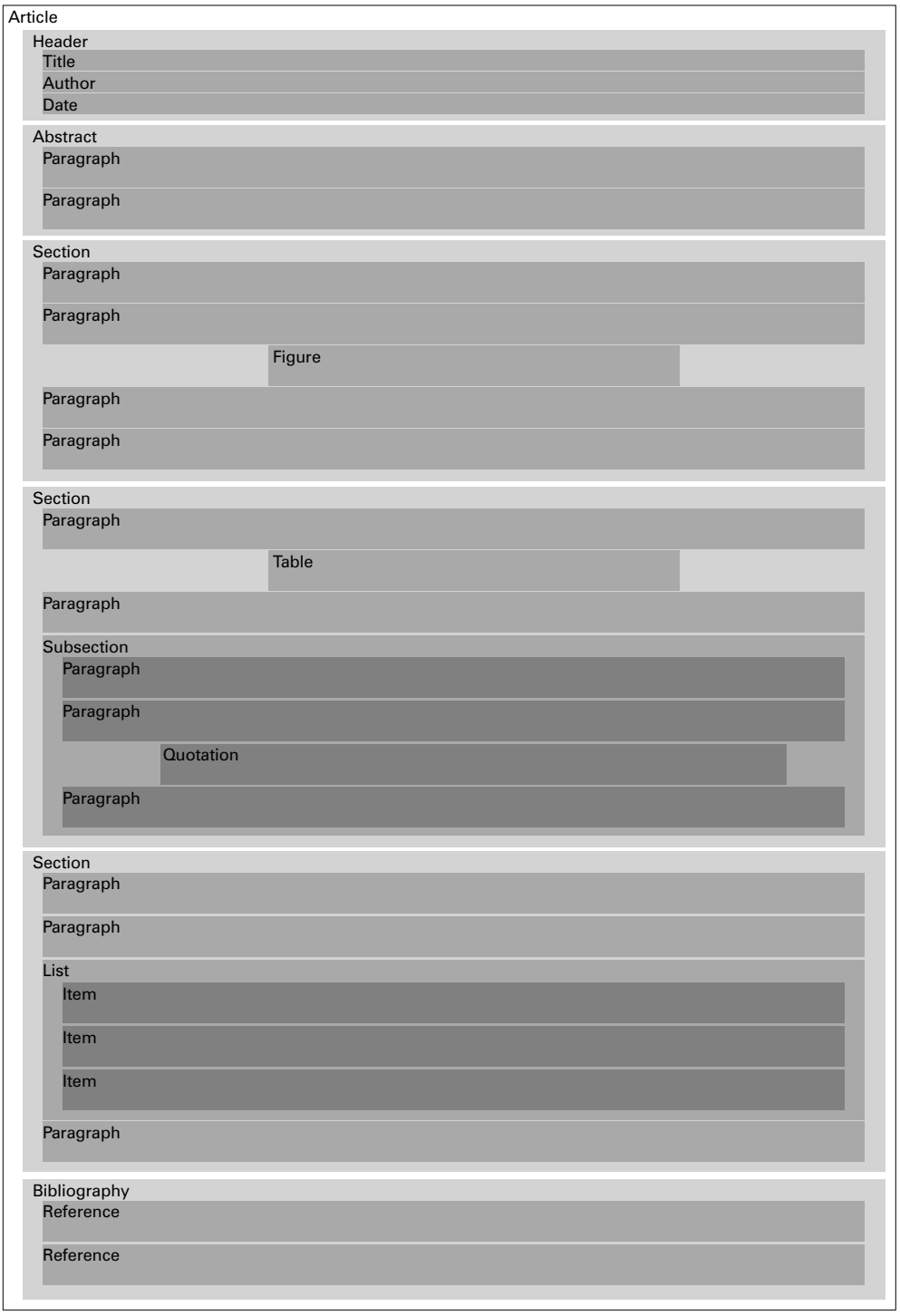
Pool: Paragraphs, block quotations, 'display' mathematics, lists, tables, figures, sidebars…

These occur and reoccur only *inside* the hierarchical elements above (you cannot have an unattached paragraph between two chapters, or an isolated list item between a preface and a foreword).

Flow: Citations, cross-references, emphases, foreign words, special terms, glosses, scientific names, fragmentary quotations, footnotes/endnotes, marginalia…

These units occur only *within those hierarchy or pool elements which themselves contain text*. In the case of notes they may be displaced to the end or the side during formatting and layout, but their point of attachment remains in the text.

The terms 'pool' and 'flow' are taken from the design conventions of Document Type Descriptions (DTDs) as used in SGML and XML (Maler & el Andaloussi, 1999), whose authors derive them from an Open Software Foundation (OSF) DTD design committee. They are in widespread use and occur in the specifications for both DocBook (Walsh & Muellner, 1999, Chapter 5) and HTML (Raggett, Le Hors, & Jacobs, 1999, Chapter 21), although they appear much earlier under the terms 'hierarchy', 'containment', and 'sequence' in Southall (1989, section 3).



Diagrammatic representation of an Article showing the hierarchical structure (header, abstract, sections, subsections, bibliography) containing pool elements (paragraph, figure, table, list, item, quotation, reference).

Figure 1.7: Components of a document expressed as 'containers'

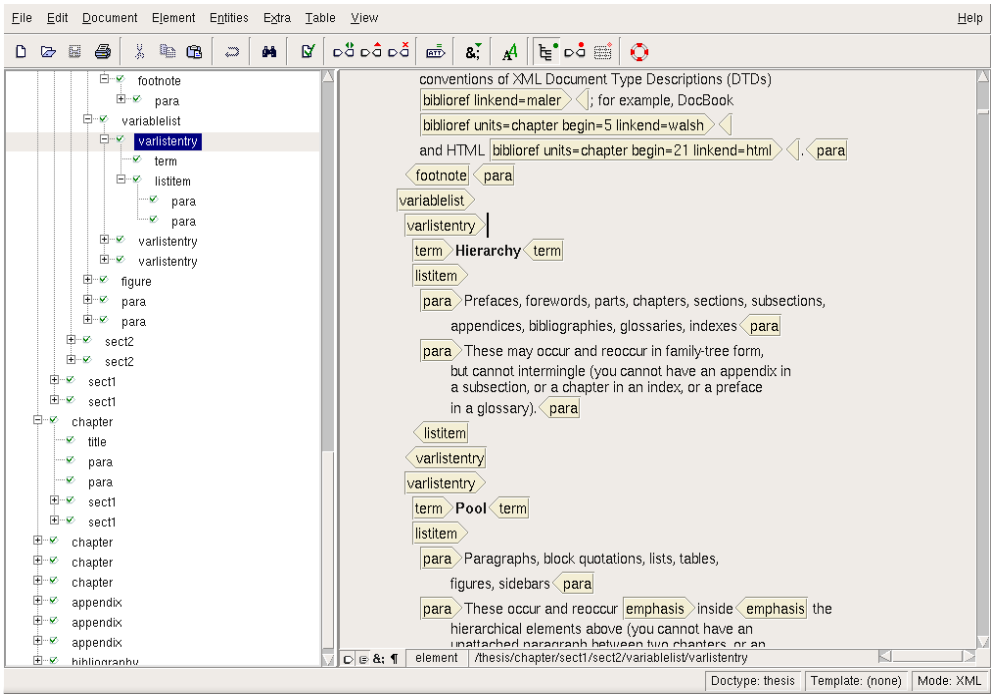
The hierarchy and pool together correspond to the 'zones' which Taghva et al. (1995, p. 320) define as 'blocks of page objects'; in XML terms they are 'elements in element content' (ie *they never occur within text*, and only pool elements may *contain* text); in LaTeX they correspond to **environments**. Abolhassani et al. (2003, p. 159) refer to this as 'macro-level' markup and define it as 'logical structure [. . .] down to the paragraph level', but it, too, makes an earlier appearance in Southall (1984), who attributes the term to Waller (1980). (Abolhassani et al. also define 'micro-level' markup as 'flow', but Taghva et al., being concerned with Optical Character Recognition (OCR), concentrate on fonts, word geometry, and white-space, rather than markup *per se*.)

Pool and flow also closely parallel the concepts of 'superordinate and subordinate categories' (the vertical dimension) and 'relations of opposition' (horizontal dimension) in the components of categorisation in HCI (Kirakowski & Corbett, 1990, p. 47). Figure 1.7 on the previous page illustrates the hierarchy and pool as a sequence of containers (compared with its expression as a tree in the next section).

1.1.3.2 The tree

The 'family-tree' or 'organisation-chart' representation of a hierarchy has become a conventional method of expressing the relationships between the components in any structural relationship, and has become formalised as part of the standard notation in graph theory and the mathematics of computing science (Knuth, 1997, p. 309). It is also familiar to most computer users from the conventional method of representing a standard hierarchical directory system, and will be recognised by writers who use an outliner for modelling their document structure (even if they are not consciously using a structured-document editor: see section 2.2.1.4 on page 81). The navigation method using boxed + and − signs to open and close the child layers is common to almost all these implementations (see Figure 1.8 on the next page).

In the formal terms of document engineering and computing science, *everything* in a structured document is a part of this tree; a strict hierarchy — in even stricter mathematical terms, an 'acyclic directed graph' (Sperberg-McQueen & Huitfeldt, 1999) — which has become a key concept in the automated processing of structured documents. The navigational terminology has been adopted from the field of genealogy (parent, child, ancestor, sibling, descendant) as well as from the



A part of this document in an XML editor, showing the representation of structure as a tree in the panel to the left.

Figure 1.8: Components of a document expressed as a tree

natural tree metaphor (branch, leaf, twig, forest, grove) and the formal language of graph theory (terminal, non-terminal).

The application of tree theory to text documents is formally expressed as an 'Ordered Hierarchy of Content Objects (OHCO)' in Renear, Mylonas, and Durand (1996), but the exceptions and objections to the rigour of this conformity have led to a growing acceptance that is it not universally applicable to all documents — the issues were well-known before, having been identified by Borkin and Prager (1981) and by Furuta (1992, p. 27).

As we explained in section 1.1.3.1 on page 16, the (OHCO) approach is also not always a useful concept for the writer. OHCO is predicated on the use of a strict tree-like document hierarchy, but real-world documents sometimes cannot easily adhere to such a rigorous structure. One notable problem is the need for **overlap**, where a textual feature starts inside one element and ends inside another: an example would be a quotation in direct speech split over several lines of dramatic verse Sperberg-McQueen and Huitfeldt (1999) or the discontinuities examined by Tazi (1989). Conventional markup practice is to enclose each line of verse in its own element, but this precludes a structure such as the following, where the

second q element starts inside the first line but ends inside the second:

```
<l><q>‘Tis some visitor,</q> I muttered,
   <q>tapping at my chamber door -</l>
<l>Only this, and nothing more.</q></l>
```

This is not well-formed XML, where all element markup must nest properly inside its parent. Criticisms of this approach have been made outside the field of literary and linguistic computing (Rusty Harold, 2007, p. 60); and OHCO has been challenged more recently (albeit very unevenly) in relation to its suitability *at all* for the large class of heritage documents in extensive use in Humanities research (Schmidt, 2010). This problem, referred to as just **overlap**, has been examined exhaustively for many years. For a useful summary, the interested reader is referred to the notes on presentations at the International Workshop on Markup of Overlapping Structures (Extreme Markup Conference 2007, Montréal) at `http://www.mulberrytech.com/overlap/`, in particular the solutions offered in Sperberg-McQueen (2007a), Sperberg-McQueen (2007b), and DeRose (2007).

However, the OHCO approach *is* applicable to the large class of regularly-formed recent and contemporary documents outside the field of historical literature. While our daily diet of reports, books, and articles may not rank as great literature, these types of document must surely be more numerous by several orders of magnitude. The benefits that OHCO brings to the handling of these documents far outweigh the problems of the relatively rare overall need for overlapping markup, which can in any case be handled adequately, if inelegantly, by a number of methods including those just cited above.

We will, however, need to consider the problems raised by using an unstructured model of the document, which we will call the ‘arbitrary’ model. This is dealt with in more detail in section 6.3.9 on page 353.

1.1.3.3 ‘Document’ and ‘data’ models

During the development of XML in 1995–96, a proposal from Microsoft representatives on the W3C XML development team put forward an approach for representing rectangular data[3] in XML form. The concept of applying SGML-style markup to tabular data was well-established in many Tables DTDs, but its use as a

[3]That is, information which is not normal continuous text, such as database records, tables, financial listings, statistical data, even log file data and configuration files.

document type in its own right for encoding data for transfer between systems appears only to have been considered rarely if at all. The formalism of this approach led to the widespread adoption of XML for the exchange of data between systems where the requirement for human readability approaches zero.

'Data' XML typically has no mixed content models, and the character data content of elements is typically numeric or categorical, or a textual label rather than narrative text. Numeric and categorical data is also extensively used in attributes, because of the controlled vocabularies that can be imposed using token lists. The XML-as-data approach led to the development of the W3C's Schema recommendation for the control of document formation, as an alternative to traditional DTDs. In W3C Schemas, stronger data typing is possible, and even character data content can be validated using a controlled vocabulary. A W3C Schema is expressed in normal XML Instance Syntax, rather than the Declaration Syntax used for DTDs, which simplifies programming, as there is only one syntax to deal with. Such schemas are typically extremely verbose, and very much larger than the more compact Declaration Syntax equivalent, but this is less relevant as they are not typically for human reading. Data XML of its nature tends to be machine-created and machine-consumed, the output of one program writing data for use by another program. In particular, where database information is concerned, the element type names and attribute names may reflect those used in the database schema, and thus both the W3C Schema *and* the XML 'document' can be generated at the same time.

'Document' XML is the traditional publishing format, as used for many years in SGML applications such as HTML, DocBook, and the Text Encoding Initiative (TEI); in the publishing industry's own formats such as the American Publishers' Association (APA) DTDs and the International Organization for Standardization (ISO) 12083 DTD; in individual publishers' in-house DTDs from Elsevier, Kluwer, Springer, and others; and in the many industrial DTDs (Flynn, 1998). Mixed content is pervasive throughout these formats, as the character data content is largely continuous narrative, and the requirement for intensive strongly-typed data values is almost entirely absent. This is the form of XML with which this research is concerned.

1.1.3.4 Free and Open Source Software

Commercial software has to be bought, and once bought, it cannot legally be copied, modified, shared, given away, or otherwise distributed without contractual rights and further payment.

But there is a large amount of software available which is free of restrictions on copying, sharing, and modification, from simple stand-alone programs to entire operating systems. Many terms have been used to describe this, including 'public domain', 'free software', and 'open source', but the matter is confused in English by the use of the word 'free' to mean both 'at liberty' and 'without payment'. The French term '*software libre*' is more exact, as it carries no connotations of cost, and for this reason we will use the term Free, *Libre*, and Open-Source Software (FLOSS). The key point in all aspects is that the user is free to modify the software, to re-use it within other programs, publish it, copy it, and distribute it. It also happens that the software is usually free of charge. It is commonly held that such software is completely free of restrictions. While this may be relatively true compared with commercial software, the popular FLOSS licences tend to have a few general restrictions: *a*) you are not allowed to prevent other people from using it; *b*) you must not try to pass it off as your own; and *c*) if you use it in a software system you are creating, then the licence for that system must also have the same terms. There are many variations on this theme, some less restrictive and some more so, but the success of the Free Software Foundation (FSF) in securing convictions for breach of the GNU Public License (GPL) testifies to its effectiveness.

Many of the most common utilities for working with XML and LaTeX are FLOSS, as is a large part of the LaTeX system (strictly speaking, the TeX program itself is not, as it predates the modern Open Source movement).[4] There can be little doubt that the generosity of spirit among software authors in making their programs available in this form has contributed significantly to the development of both systems.

1.1.4 Markup theory and practice

As we saw in section 1.1.2 on page 8, the essential function of markup systems is to allow the author or editor to indicate the important features of the document

[4] Notwithstanding this, there are several excellent commercial implementations of TeX.

so that they can be recognised, both by readers (visually-based) and by computer programs (logic-based).

There are several taxonomies of markup which have been discussed in detail elsewhere (Lamport, 1986; Coombs et al., 1987; Cournane, 1998), resulting in classifications along several axes, but for the purposes of this study we are restricting ourselves largely to the two major groups identified by Lamport (1986) which he calls 'visual' and 'logical'; the distinction is based on earlier work by Reid (1980a) and Roberts (1979) which we discuss in section 2.3 on page 91.

Visual markup: this is designed solely to *display* a given style of formatting or layout; for example bold or italic type, a particular typeface or size of font, or a certain measure of space. In such systems the onus of recognition and interpretation is placed exclusively on the reader, who is expected to be able to deduce from perception, experience, and the prevailing conventions why the text has been formatted in this manner, and what significance it has for meaning or understanding. There is no indication — least of all in the document file itself — of any of the reasons for such formatting: the only information that gets stored describes the visual effect. Visual markup is the predominant method used by wordprocessing systems.

Logical markup: this is the reverse of visual markup. In logic-based systems, descriptive marks are placed inside the document, identifying what role each piece of text plays in the document. While these marks can be represented inside an editing program in any convenient form for the programmer, they often have an external, human-readable form that can be seen in a plaintext view of the document, usually in the form of a self-explanatory label such as Title, Section Head, Emphasis, List Item, Cited Work, or Extended Quotation or (less clearly) an abbreviation or mnemonic such as TTL, SECHD, EM, LI, CIT, or QUOT. No formatting of any kind is stored in the document text itself, so a separate list of styling details (a **stylesheet** or **template**) is required which enables the computer to display the correct layout and formatting (this may also be embedded into the saved document file rather than stored separately). Logical markup is used in almost all XML and most LaTeX systems.

While these terms have been presented as polar opposites, other terms are in use which describe related models of markup, including:

- 'structural' markup (concerning the hierarchy and pool);

- 'semantic' markup (concerning the flow);
- 'implicit' markup (deducible from context, like quotation marks surrounding a quote, centering for a heading, or a new page (form-feed character) before a chapter heading, but without using tags to say so);
- 'explicit' markup (markup that means what it says, either visual or logical, such as 'italic' or 'footnote');
- 'descriptive' markup (similar to explicit markup but restricted to logical systems);
- 'procedural' markup (similar to visual markup, but exposing the programming required to achieve the result).

Many of these have overlaps with other definitions, and there are also hybrid systems exhibiting features of several different approaches, but for the present purposes the Logical/Visual distinction will suffice.

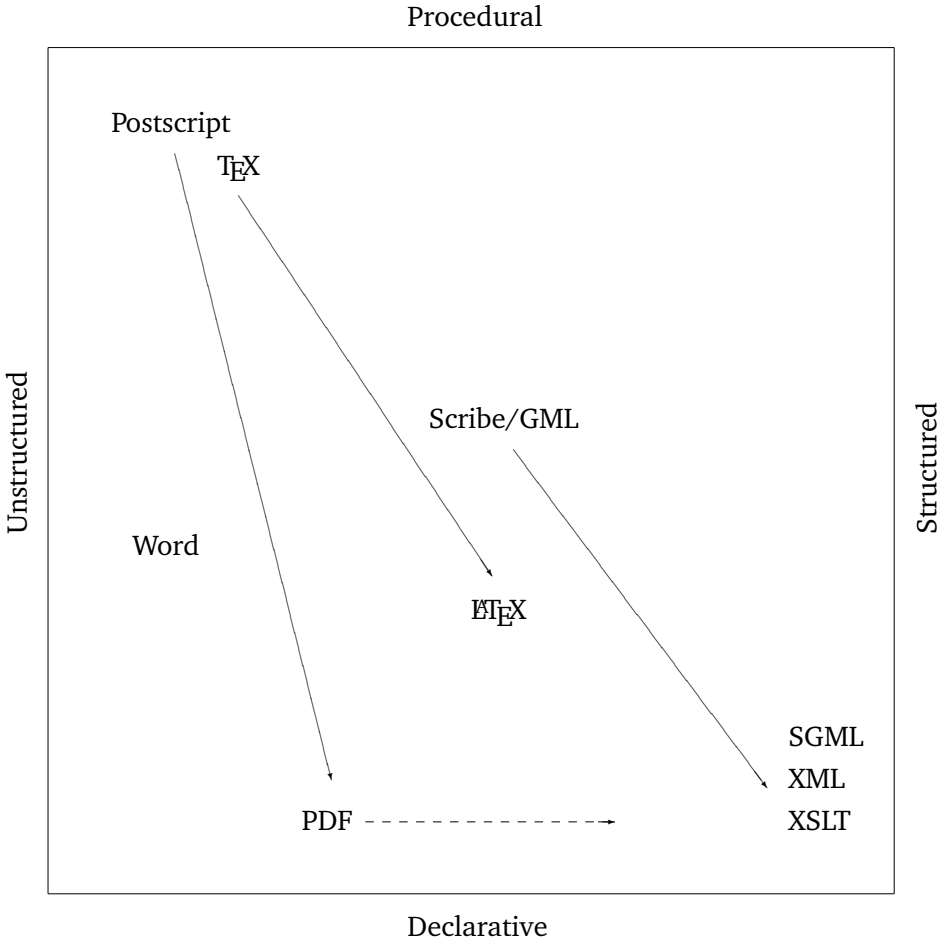
It may readily be seen that the majority of wordprocessors fall into the category of visual markup systems, because they allow the author to create visual effects whose implications can only be recognised by another human reader. Logical markup systems are less common (for reasons which are the material of this research), with the exception of HTML, perhaps the most widespread of all, and in itself something of a hybrid; and more recently, 'markdown' systems, closely related to Wiki markup, which aim at simplification and a minimalist approach.

Logic-based formatting is also widely available in wordprocessors, although it requires an additional programming language (such as *VBScript* for *Word*) to provide a means of altering the wordprocessor's behaviour: numerous plugins for *Word* have been written to enable XML editing (see section 6.4.2 on page 365 for an example). For reasons which we shall examine in Chapter 2, the method of using stylesheets with wordprocessors in this way is not in widespread use.

The forms of logical markup most commonly encountered are

- The Extensible Markup Language (XML) (Bray, Paoli, Sperberg-McQueen, & Maler, 2000), which is based on ISO 8879:1985: 'Standard Generalized Markup Language' (ISO JTC 1/SC 34, 1985)[5]

[5]We consider here XML instead of SGML. XML is a proper subset of SGML, simplified for ease of implementation on the Web, and now far outstrips SGML in terms of popularity. SGML is still in use in many places, but what we have to say about XML can be taken to apply to SGML as well, except where specifically noted.



After Mackichan (2007, p.1). *Word* is an anomaly: the file format is XML, and therefore highly structured; but this structured syntax is used to represent a completely unstructured document model.

Figure 1.9: Relationship of markup categories

- LaTeX, a document preparation system (Lamport, 1986, 1994), based on the TeX typesetting engine (Knuth, 1986).

Both evolved at around the same time but entirely independently; both principal authors have confirmed that they were largely unaware of each others' work until later. They nevertheless show marked similarities in concept, having derived in part from common earlier work such as Reid (1980a) and others (see, for example, the short historical notes in Goldfarb (2003) and Cournane (1998)). They are, however, significantly different in construction, although they share some common principles.

The following sections provide a very brief and non-rigorous explanation of the markup models implemented by XML, LaTeX, and wordprocessors.

1.1.4.1 XML

XML is formally an 'application profile' of the Standard Generalized Markup Language (SGML), ISO 8879, and inherits its syntax and strict hierarchy, but omits many of the arcane and hard-to-program options which militated against SGML's acceptance.

(The original HyperText Markup Language (HTML) used for the Web is defined in SGML, although Web browsers largely ignore the rules. The modern XML version of HTML is the Extensible HyperText Markup Language (XHTML), again designed to bring formal methods to Web browsers, again with only partial success; the latest version, HTML5, retains the syntax but has laxer rules.)

Despite its name, the Extensible Markup Language is not strictly a markup language but a markup *definition* language; that is, it does not itself define any concrete markup but provides a **metalanguage** for doing so. XML thus allows document engineers to describe the structure of each different **document type** (articles, books, reports, database downloads, web pages, newsfeeds, and thousands of others), and to name their components. These Document Type Descriptions (DTDs) contain the specifications of the markup that can be used for instances of documents of that type.

As noted earlier, there are other forms of expressing the structure of a document type apart from the DTD, known as **schemas**, of which the WorldWideWeb Consortium's (W3C's) schema language (Fallside & Walmsley, 2004) and James Clarke's RelaxNG (RNG) are the most common (ISO/IEC 19757-2:2003, 2003). There is also the validation language *Schematron*, which provides many of the same features but allows a range of programmable constraints on the grammar which are not available in the schema languages themselves. While these all provide many different features, for the conceptual purposes of this research the terms 'DTD' and 'schema' (W3C, RNG, or *Schematron*) should be regarded as equivalent, in that they provide a formal rules-based guide to the formation and validation of an XML document.

A DTD specifies rules for the identity of every generic component or **element type** needed in the type of document being described (see the panel 'Example of an XML DTD: a definition for a recipe' on the next page). The editing software reads this specification and uses it to control whereabouts in the document the various element types may occur.

In a recipe book, a recipe needs a title, an optional comment, a list of ingredients, a description of how to cook it, and possibly a reference to where it came from. This would be declared in XML as follows (the question mark denotes optionality):

```
<!ELEMENT recipe (title,comment?,ingredients,
                  method,source?)>
```

The title, comment, and source just contain plain text (in markup terminology, Parsed Character Data (PCDATA)):

```
<!ELEMENT title (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
<!ELEMENT source (#PCDATA)>
```

The ingredients element is a list of one or more items (plurality denoted by the + sign), with the quantity and units kept in separate attributes:

```
<!ELEMENT ingredients (ingredient+)>
<!ELEMENT ingredient (#PCDATA)>
<!ATTLIST ingredient quantity CDATA #REQUIRED
          units (g|Kg|l|ml|dl|oz|lb|pt|cup) #REQUIRED>
```

The method might just be a paragraph of description, or it could be a list of action items, depending on the recipe (alternation denoted by the vertical bar):

```
<!ELEMENT method (para|list)>
<!ELEMENT para (#PCDATA)>
<!ELEMENT list (item+)>
<!ELEMENT item (#PCDATA)>
```

An example of this DTD in actual use in an XML document is shown in Figure 1.10 on the next page.

Panel 1.1: Example of an XML DTD: a definition for a recipe

In practice there would be much more not given here: scope for longer or more complex comments and sources, more predefined units (and provision for their absence in the case of unitary items like eggs), and ways to allow for illustrations, cross-references, and other features.

Rules like keeping the list of ingredients separate from the description of the method are statements of the obvious for a human, but are necessary to allow the computer to govern the formation of the document in a controllable manner.

It should be pointed out, as observed in the fourth item on page 10, that both

```
<!DOCTYPE recipe SYSTEM "recipe.dtd">
<recipe>
  <title>Chocolate fudge</title>
  <comment>My mother's recipe</comment>
  <ingredients>
    <ingredient quantity="1" units="lb">sugar</ingredient>
    <ingredient quantity="4" units="oz">chocolate</ingredient>
    <ingredient quantity="½" units="pt">cream</ingredient>
    <ingredient quantity="1" units="oz">butter</ingredient>
  </ingredients>
  <method>
    <list>
      <item>Mix the ingredients in a pan</item>
      <item>Heat to 234°F, stirring constantly</item>
      <item>Pour into greased flat tin</item>
      <item>Allow to cool before cutting</item>
    </list>
  </method>
  <source>Adapted from the Good Housekeepings cookbook</source>
</recipe>
```

The declarations stored in the DTD file can be used for any recipe. In this example, the editing software has used the DTD to control what element types the author can use, and where they belong, making sure that the final document conforms to the rules.

Figure 1.10: Example of XML: the recipe definition in use

static and dynamic components of a document can be generated automatically, once the markup identifies the information accurately. Hence in our example, although there is no declared classification of the recipe (meat cookery, desserts, etc), this could exist in a larger system defining a whole book of recipes, and inferred from an individual recipe's position within the chapters. Similarly, the index and references could be generated automatically from information supplied in the title and the sources. XML files can be processed to create many types of output such as web pages, typeset PDFs for printing, audio commentary, Braille, or inputs to further processes.

Because the DTD is computer-readable, an editing program can use it to monitor and control the formation of the document while it is being written or edited, to ensure nothing is misplaced, and that the document conforms to the rules. Similarly, before processing (which may happen away from the author's computer), the entire document can be checked for conformance (a process called 'validation'). The immediate effect of this can be seen by opening a new XML document in an XML editor and specifying a DTD. In Figure 1.11 on the facing

page, the *Emacs* editor has been used in this way, and it has worked out what element types belong where for a book to be written using the DocBook DTD. It has displayed empty elements for the title, author, introduction, and the first chapter, ready for the author to start writing, with a comment inserted as a reminder in the bibliography element that the author must pick one of the allowable named subelement types, and the cursor left ready in the first element known to require text input (`title`).

```
<!DOCTYPE book SYSTEM "docbook.dtd">
<book>
  <info>
    <title></title>
    <author>
      <personname></personname>
    </author>
  </info>
  <preface>
    <title></title>
    <para></para>
  </preface>
  <chapter id="">
    <title></title>
    <para></para>
  </chapter>
  <bibliography>
    <!-- one of (bibliodiv bibliomixed biblioentry) -->
  </bibliography>
</book>
```

Figure 1.11: Required elements being added automatically by an editor

In fact, a DTD or Schema is normally only useful for controlling the formation of a document in an editor (or under program control). Once the document is finished, and is known to be valid, the DTD can often be dispensed with entirely for subsequent processing ('DTDless' processing). This is contrast to SGML, where the DTD is always compulsory. However, a DTD or Schema may still be required at this stage for several reasons:

- if it contains default values for attributes which would otherwise not be knowable;
- if there are validation criteria specified in the Schema;
- if there are any **entity declarations** (a form of expansion or external file

reference) in the DTD which are used in the document;

- if the processor is going to modify the document, in which case the same constraints apply as for formation.

Physically, the markup is implemented in XML by the insertion of **tags** into the text to enclose each component of the document. Each component thus enclosed is called an **element**, and consists of a start-tag at the start and an end-tag at the end, with **content** (text or more markup) in between. Each of the two tags is made up of angle brackets containing the name of the type of element, and the end-tag is differentiated from the start-tag by the name being prefixed with a slash. Start-tags (but not end-tags) may include additional information in the form of `name="value"` pairs called **attributes**, often used to supply classifications, identity, or other metadata (see the chapter element in Figure 1.11 on the previous page, where the `id` attribute has been provided to give the chapter a unique ID for cross-referencing purposes).

As can be seen even from the minimal example in Figure 1.11 on the preceding page, XML elements need not themselves contain text directly: they are frequently used to hold other combinations of sub-elements — in the example, the chapter element contains no text itself, only the title element, paragraphs, lists, and others. The content of such elements is referred to as **element content**. By contrast, the title, paragraphs, and other elements which contain actual text mixed with further sub-elements are referred to as having **mixed content**. We will be discussing the effect of these two types of content on editing interfaces later.

In practice, a synchronous typographical XML editor for the conventional author will not be expected to display the markup. This has advantages and disadvantages for usability which are precisely the topic of this research, and which we examine more closely in section 3.2 on page 133. In Synchronous Typographical (ST) editors, knowing what element your current cursor position is in can be difficult when the element is newly created, because it contains no text, and therefore shows no evidence on the screen of its existence. Some software provides a marching ‘tell-tale’ at the bottom of the screen, showing the name of the current element at the cursor location.

1.1.4.2 LaTeX

LaTeX is a document preparation system, designed as a set of **macros** (subprograms or reusable program fragments) for the TeX programmable typesetting language. It is intended to allow the TeX typesetter to be used for formatting without the author or editor having to know the programming details necessary to make it work.

The underlying TeX program (unseen by most authors) implements an extensible set of about 300 procedural formatting commands known as **primitives**. LaTeX, however, defines mostly structural markup, with a goal similar to that of XML, but without the strict enforcement of a particular structure. It also allows the manual inclusion of arbitrary visual markup so that the expert typesetter or programmer can optimise the appearance, but this is deprecated for the conventional author because of the existence of a very large number of add-on 'packages' (combined markup and style controls) which obviate most manual adjustments.[6]

For some structural controls, LaTeX uses a 'container' hierarchy similar to XML, with `\begin` and `\end` commands. On the other hand, chapters, sections, and subsections are marked simply by the existence of their titles, so that one division is deemed to have ended by virtue of the fact that a new one has started.[7] However, lists, quotations, figures, tables, and many other components of a document are marked with a beginning and an end.

There is no structural equivalent in LaTeX to the XML DTD. The term **document class** is used for a file which specifies what features (commands) are expected or available in a given document type, but it does not enforce their use in the way that a DTD does for XML. In the default book and report document classes, for example, the command `\chapter` is defined, whereas in the article class it is not, as chapters do not occur in articles.

Figure 1.12 on the next page shows the equivalent empty LaTeX document to that shown for XML in Figure 1.11 on page 31. Comparison between these two will readily show the strong similarity, with the locations for title and author, the

[6] As of October 2012, there were 4,355 packages listed at `http://ctan.org/pkg/`.

[7] SGML, as distinct from XML, also offers the feature of a missing end-tag, whose effect can be inferred from the presence of the start-tag of the next element; but for a different reason: the minimisation of file space, and of effort in the days when all markup had to be typed manually by the author. In the case of LaTeX, there are processing efficiencies to be gained from allowing formatting styles to be global, rather than constrained within a container (Stephani, 2009).

```
\documentclass[a4paper,10pt]{book}
\begin{document}
\title{}
\author{}
\maketitle
\section*{}

\chapter{}\label{}

\bibliography{}
\bibliographystyle{}
\end{document}
```

Figure 1.12: A skeleton document created in a LaTeX editor. Note the strong structural resemblance to Figure 1.11 on page 31

introduction, first chapter (with provision for the cross-reference label, which fulfils the role of an XML ID), and the pre-set bibliography style.

Although it is possible for an XML author to redefine the markup in the DTD, it requires significant programming skills; in LaTeX, the author can arbitrarily redefine the names and meanings of the markup with less foreknowledge required, although there are syntactic traps for the unwary. For consistency, this kind of modification should obviously not be undertaken lightly, but it has become a hallmark of LaTeX documents submitted to publishers that authors are prone to unnecessarily reinventing their own wheels with excessive frequency: we discuss this tendency further in Chapter 2.

The physical markup is formed of **control sequences** (better, if less accurately, known as **commands**) consisting of a backslash followed by the name of the command, and followed where necessary by **arguments** in curly braces. The argument or arguments, where needed, usually consist of the text to act upon (such as a chapter title) or optional settings (in square brackets, such as in the first line of Figure 1.12). Normal paragraphs have no explicit markup, but are delimited simply by a blank line.

The containment effect is achieved by **environments** bounded by two special commands, `\begin` and `\end`, followed in curly braces by the name of the container. The example in Figure 1.12 shows this used for the outermost container environment called *document*.

There is no full validation process akin to that used in XML, except the running of

the TEX program itself (the act of typesetting), although informal syntax-checkers exist in many editing programs to catch the most common errors such as mismatched braces or mis-spelled command names. Because of the flexibility of redefinition mentioned above, there can be no formal statement of what is permitted or denied along the lines of a DTD or Schema, except the macro definitions of LATEX itself, and even those may be overridden by the expert. There is, as mentioned earlier, a very large collection of additional preprogrammed material (**packages**) which implement a wide range of common typesetting requirements using the standard `\begin...\end` or simple command syntax. Over the years this has substantially lessened the need for the author to reinvent the wheel.

1.1.4.3 Wordprocessor formats

Despite their pervasive presence, wordprocessor programs do not provide a robust key to document structure, either in their interface or their file format. They generally lack both depth and containment beyond a single level; that is, everything is a paragraph, and only the formatting or a style naming system indicates the role of each 'paragraph' in the document.

However, both major systems (*OpenOffice* and Microsoft *Word*) save their documents in an XML format by default, although this simply represents the visual appearance, and makes no attempt to impose or reflect a document structure.[8] While it is therefore possible to process such documents with an XML processing language such as Extensible Stylesheet Language (Transformation) (XSLT) or *Omnimark*, the hierarchical position or structural 'ownership' of a sequence of elements can only be detected by 'looking ahead' to the next occurrence of a structural marker in order to locate the bounds of the current division. This is exceptionally time-consuming to program, and computationally expensive to process, even with the help of style names, but the method has been used successfully in some conversion products, notably *DynaTag* (DeRose & Vogel, 1995), still available from Red Bridge Interactive, successors to its creators; and more recently in experimental software such as *Adaptive READ* (Drawehn, Altenhofen, Stanišić-Petrović, & Weisbecker, 2004).

For these reasons, wordprocessors cannot be considered as structured document editors *per se*. While *WordPerfect* (and more recently, *Word*) do have a full XML

[8] With one exception: *OpenOffice* does enclose its lists in a list container.

editing mode of their own, this is entirely separate from their normal wordprocessing mode. These specific variants were initially evaluated with the other software in section 3.2 on page 133 but later removed from the analysis.

1.1.5 Representation

Depending on the circumstances, the representation of the material may be left entirely to the writer's discretion (for example, a well-known and trusted author will have great latitude and encounter little control from a publisher); or it may be tightly prescribed and admit of no variation (for example, pharmaceutical safety documentation, which is rigidly controlled by law), or — in the majority of cases — it may be somewhere between the two extremes.

1. writers choose a particular physical form to convey or represent their thoughts (a whole chapter, or a list, or a poem, or a sequence of paragraphs, or some other structure) in a way which will be understood by the reader;
2. they then choose a style or format which instantiates that form in the editor or wordprocessor; that is, their mental model of how to express those thoughts is now given a concrete representation in the computer;
3. the editing software uses styling to represent the words as they are typed, in the format selected (or as modified by a human editor, who may re-cast the format for technical or other reasons); but the software represents them in memory in one way while still being typed or edited, and on disk in another way (for example, XML) when the document is saved;
4. when the finished document is published (on the web, in print, in an Ebook, as an audio file, etc), the markup is taken as representative of the writer's intent, and transformed to the appropriate new markup required;
5. the rendering engine (browser, typesetter, reader) represents this markup as patterns of dots or lines or sound waves, taken from examples (fonts, speech samples) which are themselves represented in yet other forms of markup;
6. the end reader absorbs these patterns with the eyes or ears, and the brain interprets the patterns by means which are outside the scope of this research.

A change in type style therefore indicates that the word or phrase possesses some special quality which the author wishes the reader to notice. In WYSIWYG mode,

the markup is implicit because the nature of that quality has to be inferred from the context, the conventions of the discipline, and the reader's experience. There are at least nine reasons why an author or editor might want to use italics (Flynn, 2002, Chapter 8), to which we have added two more at the end, taken from Walker (1983) which we discuss in more detail in section 2.2.2 on page 85:

1. foreign words;
2. scientific names;
3. emphasis;
4. titles of documents;
5. names of products;
6. mathematical variables;
7. headings;
8. letters used as words;
9. decoration;
10. distinction/particularisation;
11. editorial intervention.

The difference between emphasis and distinction or particularisation is that emphasis refers to a word or phrase within a paragraph, intended to highlight the fact that it has special weight; distinction or particularisation usually refers to a whole paragraph or other block of text italicised to set it apart from the material before and after (this use is often didactic or tutorial). In item 11 above we subsume all the many reasons for italics used in transcriptions and other editorial matter, for which the TEI provides ample markup to represent correction, elision, abbreviation, quotation, citation, and interpolation, to name but a few. There are doubtless others, and many more related to other type sizes, styles, and weights (bold, underlined, sans-serif, script, etc).

A few of these we encounter as children, soon after learning to read. By adulthood the reader may subconsciously interpret many of them, and is able to elide without difficulty the reason for the italics by their placement in the text. Southall (1989) refers to this as 'an interpretation of the typographic structure based on precise rules' (Southall, 1989, p. 139), although some might query the precision: humans are very good at 'fuzzy' reasoning, and may recognise features even when disguised by accident or design as something else. Some features remain highly domain-specific (Linnaean names, for example, or mathematics), and while the reader may notice them, understanding their significance requires the specialist knowledge of the domain.

1.1.6 Tag abuse

The term **tag abuse** is used to describe what happens when an author or markup editor has a requirement to identify a particular word or passage in a document, but fails to find suitable markup to describe it. Typically there are three courses of action possible:

- mark it up with the closest available match, if there is one, and document the action and reasoning;
- submit a formal request for a new element type or attribute designed for the purpose (perhaps in conjunction with the previous action);
- 'borrow' an element type originally designed for something completely different, but which experience shows will end up looking the same in the final document rendering.

This third action is what we term **tag abuse**. The term was originally intended to be humorously derogatory, aimed at highlighting either the author's lack of understanding of the markup, or the document type designer's failure to provide for the case in question. However, some specific classes of tag abuse have become so common that it has become the standard term.

While tag abuse must certainly have occurred in all schemas at all levels, the *loci canonici* relate to HTML and LaTeX: both vocabularies provide markup for emphasis, and both provide markup for undistinguished italics.

Vocabulary	Emphasis (Logical)	Italics (Visual)	Rendering
HTML	`you <em>must</em>`	`you <i>must</i>`	you *must*
LaTeX	`you \emph{mustn't}`	`you \textit{mustn't}`	you *mustn't*

Novice users, or users accustomed only to wordprocessor controls, may not be aware of the existence of semantic (logical) markup or of any reason why it should even exist in the first place. If they just use italics, it may be because their reading experience has led them to 'know' that emphasis is shown by italic type.

In fact, from observation, most LaTeX editors we will be investigating have an emph button or menu item clearly intended as an affordance to try and deter the user from using undistinguished italics (HTML editors may likewise provide an em button). In fact, the LaTeX command `\emph` has the feature that it produces italics when used in normal (upright) text, but when used in text that is already italic (for some other reason), it produces upright text, which is standard publishers' practice.

Possibly as a result, the semantically accurate usage (or \emph) has been emphasised so often in responses in mailing lists and newsgroups that some browbeaten users now regularly mark non-emphatic words or phrases with or \emph where it would in fact be more appropriate to use <i> or \textit. Some HTML editing software for content management systems such as *TinyMCE* and *HTMLArea* can be configured to replace <i> silently with long before the <i> became deprecated by the W3C standard — the standard is, in effect, encouraging **tag abuse**.

More egregious abuse is largely the result of the vocabulary not matching the requirements of the task, either through having been superseded by change, or from there being no proper mechanism for requesting changes (Maler & el Andaloussi, 1999),[9] or because the vocabulary was originally designed for a different class of documents, and is being used because it offers other facilities whose utility outweighs the disadvantages of having to abuse some element types.

1.1.7 Definitions

For the purposes of this research, we define a 'structured document' as a document which exhibits a conscious division and subdivision of the material into a regular pattern of reoccurring units which form an ordered hierarchy.

- No restriction is placed on the breadth or depth of the hierarchy: it may be as simple as a novel consisting of chapters containing paragraphs; or as complex as a large piece of legislation with hundreds of sections and subsections nested as many as nine levels deep.
- It is a consciously ordered division because the author or editor has done it deliberately, or has used software which has done it for them; that is, the document has had order imposed upon it.
- The units are reoccurring because each of them forms one instance of a type of lower-level 'building-block' from which the level above can be constructed.

[9] The usage of 'decoration' was added to the author's original eight possible reasons to use italics in the list on page 36 after hearing from a colleague who reported that her client had editorial staff insisting that they should be able to specify italics anywhere in the document they wanted, without any semantics attached, to make it more visually appealing — a perfectly valid motive.

There is considerable confusion even within the computing industry over a number of terms often used synonymously:

WYSIWYG: The familiar phrase 'What You See Is What You Get' (WYSIWYG) describes the relationship between the appearance of the document on the screen during creation or editing, and the appearance of the document when printed or formatted to a printable file format such as PostScript (PS) or Portable Document Format (PDF). It implies that there is a 1:1 correspondence between the two, such that anything visible on the screen will appear in the printout in an identical form.[10]

Despite common practice, WYSIWYG does *not necessarily* refer to the edit window, although this is implied when a synchronous typographic editor is in use. It is perfectly possible to use an asynchronous editor for editing the document, and a refreshable typeset preview display to provide the WYSIWYG view. For a more complete discussion of this in respect of LaTeX, see Kastrup (2002).

GUI/TUI: Graphical User Interface (GUI) refers to a class of interface in which documents, folders, programs, and other objects are represented pictorially and manipulated with a pointing device in real time.

In this interface, text is conventionally represented in a selectable printers' typeface rather than the fixed-width typewriter typeface used in console or character-cell applications, known as Textual User Interfaces (TUIs).

The term is also misapplied by extension to the programs which run within a GUI; whence the phrases 'GUI editor', 'GUI browser', etc.

Direct-intervention editing: This is a misnomer. It describes the act of editing by using a GUI to modify the typeset output within a WYSIWYG display, which in turn causes the relevant edits to be made to the underlying document.

It should of course be called 'indirect-intervention editing', but the conflation of 'document' with 'appearance' in the user's mental model has elided the distinction.

Synchronous typographical editor: A more precise description of the whole

[10] Barbara Beeton reports that the term was coined by Bill Tunnicliffe in a presentation at a 1978 committee meeting involving the Graphic Communications Association (GCA), the American Mathematical Society (AMS), and the Printing Industries of America (PIA).

> class of software that allows documents to be edited by modifying the WYSIWYG typeset display within a GUI by 'direct intervention'.
>
> It distinguishes this behaviour from editing documents by all other methods, regardless of whether these other methods also possess a WYSIWYG preview, act by direct or indirect intervention, or operate within a GUI or TUI.

All other definitions used are taken from the prevailing standards documents cited in the prefaratory section 'Terminology' on p. xxiii.

1.2 Scope of the research

The initial impetus for this work arose from a desire to find some explanation for some of the author's experiences as a technical advisor in academic computing support and industrial IT consultancy.

1.2.1 The nature of the problem

Direct involvement with computer users and their difficulties with editing software for structured documents has indicated for many years that there are some serious, persistent, and unresolved problems with the editing interfaces which prevent their efficient use. These problems initially appear to be related to the level of technical knowledge expected of non-technical writers and editors, and to the way in which the interface of the software is presented. Such observations tend to be supported by anecdotal information from colleagues in the field, but there has been a lack of empirical evidence to support or deny this view, although there is some evidence to suggest that a critically cynical approach by writers has a large part to play (Boice & Jones, 1984).

During the period 1996–2001 the technical requirements for certain types of structured document were greatly simplified as a result of the development of XML (Bray et al., 2000), the details of which are described in section 1.1.4 on page 24. Earlier systems working on the same principles had been available for over a decade, notably SGML (ISO 8879:1985, 1985) and LaTeX (Lamport, 1986); but whereas LaTeX had been (and remains) in extensive use in publishing, academia, and scientific fields, SGML was never in widespread use outside a few specific industrial and academic areas, partly because of its complexity and the high cost of the software. We discuss the evolution of these methods in more detail in section 2.2.1.2 on page 76.

It also became clear during this period that the introduction of XML meant that although the technical foreknowledge needed for editing structured documents was reducing considerably, there was no obvious change in the type of software available (see section 3.2 on page 133). This discrepancy appeared never to have been investigated in detail. In the same period there also was a noticeable increase in the number of requests in discussion forums for software to handle the new types of document made possible by the changes in technology (see section 3.3 on page 161).

1.2.2 Constraints

The scope of this investigation is therefore constrained by three major exogenous factors:

1. The nature of the software available;
2. The requirements of the users or potential users;
3. Their experiences and behaviour with existing software.

A single endogenous factor is given: the particular type or types of structured document format used, which by implication includes the underlying technologies used to create and maintain the documents. In the present study these two formats are XML and LaTeX, for reasons explained in section 1.1.4 on page 24. For operational reasons, we also accepted the limitations that we are not writing an actual editor, but testing some changes to the interface of existing software.

The investigation therefore attempts two major tasks:

1. to identify the variables which condition the three exogenous factors: this is done in section 3.2 on page 133, 'Software analysis', section 3.3 on page 161, 'Requests analysis', and section 3.4 on page 176, 'User survey';
2. to suggest changes in the software interfaces which might accommodate the users' requirements and lead to greater acceptance of the software; these are identified in Chapter 4 and tested in section 4.5 on page 283.

The user populations include *a*) those who write or edit structured documents on a regular basis; *b*) those who have occasional need to do so; and *c*) those whose employment has just required them to do so. This last category is expressly included because it was found to be a common reason given for asking for help in messages to the principal discussion forums. The target population expressly excludes professional technical writers, computing scientists, markup experts, and document engineers, all of whom can be presumed to have extensive foreknowledge of the internal details of document construction.

The software is confined to that designed explicitly for structured documents. General-purpose editing software (wordprocessors and text editors) is therefore excluded, as explained in section 1.1.4.3 on page 35, with the exception of those systems which have an additional mode of operation for handling structured documents.

Without prejudice to the investigation of author requirements, the words of one writer upon discovering markup (or rather, *Markdown*), make a useful summary:

> In my opinion there are 3 solutions to 'The Problem':
>
> 1. Better education: students should learn how to use apps in order to be productive. Computers have been designed to work fast and hassle-free.
> 2. Better integration of existing apps: eg apps should have better options built in to export the document and to create a final document. The final document matters: what if you have five awesome apps and you lose time to connect them and transfer your docs back and forth? Productivity should be productive.
> 3. Big disruption: everybody needs a powerful text software. Someone has to build a system that hits the sweet spot.
>
> (Levi, 2013)

1.3 Methodologies and tools

The range of methodologies and tools available for the investigation and testing of interface usability is briefly reviewed below. However, the current investigation concentrates specifically on the usability of editing software for structured documents. Some of the tools and methodologies for the comparison of different means of control were developed for other levels of the interface, so they do not always provide a basis for the analysis of the requirements for editing structured documents (as opposed to other kinds of document) set out in section 1.1.7 on page 39.

Some generic interface components, such as buttons or menu items for saving the current document, or controls for zooming the magnification factor of the display, are not specifically related to the editing of structured documents. Their use can be applied globally to almost any application, and their specification for their design and effectivity are included in common interface behaviour specifications (Apple Corporation, 2008; Microsoft Corporation, 1995; The GNOME Usability Project, 2008). We will not attempt here any analysis of the generic functions, as

this is covered extensively in the literature on user interfaces (see section 2.2.2 on page 85) and interface guidelines (see section 2.3.4 on page 98).

As a result of the investigations in Chapter 3 we were able to identify some design patterns which are specific to the implementation of an editing interface for structured documents, as distinct from those which are more or less generic to any interface, and these are the ones we tested.

1.3.1 Methodological development

In common with many other developing fields of study, the early period of Human-Computer Interaction (HCI) tended initially to feature the relatively isolated work of individuals in the 'parent' areas. These included seminal contributions from researchers in disciplines such as cognitive psychology, physiology, interface design, ergonomics, and computing science (Green, Payne, & van der Veer, 1983; Carroll, 1987), as well as early work done in military and industrial fields dating back to WWII.

The mature field represented by the present-day study of Usability has inherited richly from these areas and many others, but also from the emergence of a 'humanistic' or 'user-centered' approach (Norman, 1988), whose conceptual roots can be seen as far back as McGregor's *Theory Y* (1960). Current work and methodologies largely use the principles of User-Centred Design (UCD) which we discuss in more detail in section 2.1.2 on page 59

Most important for methodological development has been the move towards adoption of standard scientific method in experimentation, measurement, and testing (Kirakowski & Corbett, 1990, Chapter 4). Of particular relevance in the field of text editing has been the GOMS methodology mentioned earlier (Card, Moran, & Newell, 1986; John & Kieras, 1996): while its original premise is now long outdated, and superseded by different technologies, the underlying principles helped frame the design of the tests undertaken in section 4.5 on page 283.

1.3.2 Hybrid methodology

The default methodology is the standard scientific method, in which a phenomenon is observed and a hypothesis constructed to explain it, which is then

tested by experimentation, and accepted or rejected, subject to verification by replication.

However, given the narrow terms of the present enquiry, the multiple forms of the individual items to be investigated, and the nascent or emergent nature of the target population, both the methodology and the interpretation of any results require some additional treatment: standard scientific method has to be applied with discretion to avoid confounding the variables involved.

Narrow terms: Investigations of user interfaces tend to focus on the usability aspects of the interface components themselves, rather than on the user's interaction with the information being processed. The nature of the information does occasionally impinge on the investigation, for example when it concerns the suitability of an error message or the applicability of a control, but the underlying format of the information is not usually relevant.

However, when dealing with an interface to a structured text application, especially where the user's interaction with the document structure is an integral component of the evaluation, we must consider *both* the interface *and* the format of the information it is expected to handle. This imposes an important constraint on the scope of the examination, because a user action (for example, adding a new section to a document) cannot be judged solely in terms of the choice of widget (whether it ought to employ a drop-down menu or a toolbar icon), but also in terms of whether the action is meaningful given the user's current location in the document structure.

Multiple forms: In common with many text-handling interfaces, there are often several different ways to achieve a particular result. While it may be desirable from many viewpoints to keep the number of such parallel paths to a minimum, it may be more comfortable for the user to have the choice of doing things in a way which is appropriate to how they view the task (in computing terms, providing **redundancy**). This means that what may well be essentially an identical task may sometimes be seen as best approached by one method, and at other times by another, depending on the user's perception.

An example is the creation of a new paragraph while writing. Authors may think of this as a perceptual break in the train of thought, larger than just ending one sentence and starting a new one; sufficiently large to warrant a slightly longer pause in the flow of information. This is reflected in the

terms 'new paragraph' or 'paragraph break' commonly used for describing the phenomenon, which we discuss in more detail in section 4.3.3 on page 246. Other authors — particularly technical writers accustomed to working under constraints of extreme structural rigidity — may think of it as the completion of one unit of description, and the creation of a container for the next one. Most structured editing systems do indeed provide for both cases, with one function ('split element') for breaking the current paragraph into two, and one for inserting a new paragraph below the current one. The net result is identical, but we are introducing additional variables to the model, and this will need to be allowed for in the analysis.

Emergent population: Text editing is (in computing terms) an old phenomenon dating back over half a century. Its use in the general modern population is most common in wordprocessing systems, where markup is invisible, but increasingly in web-based systems (web sites, blogs, and wikis, where markup is visible, or at least revealable) and in mobile communications (interactive messaging, tweets, and cellphone text messaging), where formal markup is non-existent).

Structured document editing as we discuss it here is a much smaller field, until recently occupied mainly by specialists in technical documentation, experienced in working to a highly-defined pattern. The widespread accessibility of the more recent web-based developments mentioned above has resulted in interfaces providing a more casual and forgiving editing environment. This means that while users accustomed to the stringent controls of a formal document system may find themselves ill-equipped to deal with the lax controls of a wordprocessor, those with experience of the web-based systems may encounter fewer problems with the concepts of markup than many observers have hitherto assumed.

When controlling for static or background variables in an experimental situation, the differential pace of learning is an additional factor which must be taken into account.

1.3.3 Persona analysis

The use of personas has become a standard tool in the assessment and measurement of software usability (Cooper & Reimann, 2003, p. 56). The construction of mental models of user requirements has long been a stage in

interface development (Barfield, 1993, pp. 45–46), and was popularised as the 'persona' concept by Cooper (1999, p. 123 *et seq*). This concept emphasised designing for a single person, on the grounds that it is not practicable to make every product suit every possible user. This constraint was repeated in a later edition (Cooper, 2004, p. 124) but is not explicit in (Cooper & Reimann, 2003), where their use to represent *classes* of users is emphasised.

A persona is essentially a model of a representative class of user, reflecting their key requirements, habits, and approaches; Snyder (2003, p. 340) calls them 'made-up examples of typical users'. Their value is that they provide something for developers to test concepts and behaviours against — a framework within which to position examples of the people who form the target population — and this will be important when we come to examine the development and testing of an interface model (see Chapter 4).

At an early stage in the current investigation, four personas were envisaged, to represent the authorial and editorial users from the structured and unstructured viewpoints. This was based on informal discussions held with some of the participants in the Expert Survey (see section 3.1 on page 107), and was designed to enable categorisation of later test results according to the perceived differences in approach between users who create text and those who have to manipulate it afterwards; and between those users who habitually write to a formal structure and those who do not (see Figure 1.13).

	Author	**Editor**
Structured	Technical writers; those with formal training in structured document techniques; those already using structured document editing systems.	Technical editors; editors in publishing houses; those with extensive experience editing submitted work (for example, from students).
Unstructured	Copywriters; 'reluctant writers'[a]; 'creative' writers; those with no prior experience of structured document techniques.	Infrequent editors; those editing informal matter for social publication; editors of material not subject to external controls.

a. The term is borrowed from the field of music, where the 'reluctant' performer is one who, while perfectly competent, is not a practising musician, but is drafted in to fill a gap in the ranks.

Figure 1.13: Initial persona categorisation

However, during the development of the Requests Analysis (see section 3.3 on page 161), it was apparent that this top-down approach was too absolutist, and

left no room for change as further data was uncovered. In particular, allowing the framework to be governed by anecdotal data and supplemented by empirical data was considered to be the wrong way round. It was also felt informally that the grid exhibited a degree of understandable cynicism by the expert group, possibly derived from habitual professional necessity in making good the damage caused by clients' lack of awareness of the use of structured document techniques.

This framework was therefore rejected, and the personas were drafted and refined solely on the basis of the results of the research. The key aspects of this drew on the Expert Survey, which emphasised the importance of interface familiarity; and on the Requests Analysis, which highlighted those features of the interface most in demand. The personas which emerged from the Expert Survey and Requests Analysis were validated by point-comparison with the data from the User Survey (section 3.4 on page 176), which provided an opportunity to test them against reported behaviour. The Software Analysis (see section 3.2 on page 133), which had no user involvement *per se*, was useful in explaining the possible influence of perceived differences between products (when in fact the actual differences are negligible).

The distinction between author and editor, while valid in some circumstances, was felt to be less meaningful in practical applications, where authors have access to the same techniques and undertake much of the same procedures as editors do.

In drafting the personas, we were led by traits which we had observed and noted in practice as being either frequently-occurring or representative of a class of users. In particular, earlier (unrelated) revisions of XML and LaTeX training courses were based largely on information gathered from attendees about the work they undertook and the software they used.

The outline characterisations were discussed with colleagues at the TeX conference in 2008, whose suggestions for the types of documents involved were incorporated. A further revision took place on a similar basis at the Balisage (markup) conference in 2009, where the employment or occupational detail was amended to reflect particular aspects felt to be important for working with structured documents. A final revision was carried out during the building of the model of the interface which would incorporate the changes we had selected for testing. Details of this are in section 4.2.1 on page 220, with notes on the revisions described above.

Annie: works for a company that handles the CVs of very highly-qualified people, running the web site and preparing reports on their backgrounds and experience for potential employers. She is a historian by training but also a computer scientist, so she understands the concept of markup and the need for a robust structure. She's under pressure to get each job done, but the information is so widely-scattered and inconsistent that there is a lot of copy-and-paste as well as original writing.

Currently she uses *Word* with a stylesheet, because there is extensive document repurposing. She regards the provision of a familiar interface as essential.

Bruce: has his own publishing business producing specialist reports and journals in a variety of fields; mostly, but not exclusively, scientific. He arranges for experts to write articles, then edits and formats them himself for the printer and web site (his colleagues deal with sales, marketing, and finance). The incoming material is in a wide range of formats, *Word* predominating, but all requiring extensive editorial conversion and preparation.

He uses a mixture of tools, including wordprocessors, formatters, typesetters, and layout programs, but he still does most final production in an ancient version of *PageMaker* because he's most familiar with it, although he knows that he will have to change to something else sooner or later.

Carrie: is a university administrator handling study-abroad programmes and their students. She has extensive correspondence with foreign institutions as well as reports on performance, conduct, and finance, parts of which are subject to discovery under a Freedom of Information Act as public records. Her background includes an MA in English Literature, so she writes fluently but has had little or no exposure to such aids as stylesheets or document structure.

Most of her work is done using a wordprocessor-like front-end to the old and cumbersome university admin document system, or with Microsoft *Word*, both of which she finds tedious but unavoidable. She is uninterested in what goes on behind the interface so long as her output is accessible in the right place at the right time.

David: is a successful science-fiction author with a background in journalism, theatre, and IT. With half a dozen novels and numerous short stories published, he has a constant stream of articles to write as well as his current book in progress (and another two in development) and two unfinished plays on the back burner. He has written his own filing system to keep track of the myriad characters, places, objects, dates, and events across his storylines.

As a former IT worker, he is well aware of the technology, and has used a number of wordprocessors, outliners, and 'assistants', but switched a few years ago to Apple's *Pages* as his main writing tool, although his publisher requires *Word* for change-tracking. He knows that change is the only constant, and is keeping an eye on other systems, both from a writer's point of view as well as for technical interest.

Figure 1.14: Personas

1.3.4 Methodology and selection

The formal statement of our methodology follows the requirements outlined above.

1.3.4.1 Observation

As described in section 1.2 on page 42, we have observed for many years the very slow rate of adoption of structured-document techniques in fields where they would have been expected to provide significant improvements in productivity or accuracy (Ensign, 1995).

Anecdotal evidence — always the first news to reach the ears of the professional in any field — suggested from an early stage that a resistance to adoption was based on the need for a relatively deep foreknowledge of the underlying file format (XML, LaTeX, etc).

Comments from users appeared to support this view, and by extension this implied that the interfaces available exposed too much of the markup (pointy brackets or curly braces), or failed to use the interface in a way which would shield the writer. A commonly-expressed opinion of authors could be paraphrased as 'editing software ought to look like *Word*' if it is to be regarded as 'usable'.

Empirical evidence is scanty. While there is extensive research on user interfaces *per se*, and on their use for text editing in plaintext editors and style-guided wordprocessors, the topic of editing structured documents has received less attention, and the emphasis has been on the training of encoders in existing software, rather than on adapting the software to meet the needs of authors.[11]

1.3.4.2 Hypothesis

In formal terms, we hypothesise that for some set of core actions in structure-based editing, there are interface methods which minimise error and user disaffection.

The implication in usability terms is that there are ways of doing things which cause minimal disruption to the user, and yet achieve the desired result; which in

[11] Encoders are editors, analytically skilled in a particular domain, who add markup to existing documents such as transcriptions.

other circumstances are only obtained with difficulty, increasing user frustration (Norman, 2004, p. 139).

1.3.4.3 Sources of evidence

In order to test the hypothesis we first needed to measure the observed phenomenon, partly to quantify its incidence and partly to provide some qualitative background. We also needed to identify the features of the software and their utility to authors. A measure of the match between features provided and features demanded was necessary to identify core actions; and details about the users' actual behaviour while writing or editing would provide information on existing use of affordances.

The preliminary measure of observations was done with a survey of expert users, seeking their judgement on the prevalence or otherwise of the difficulties users experienced with the software. A group of experts in the field was selected by personal contact and invited to answer an administered questionnaire. This provided valuable baseline data, feedback for the subsequent User Survey, and some specific quantitative and qualitative points of reference.

The current and future axes of the human dimension were originally to be measured using three surveys (Expert, User, and Novice). However, in the light of the results of the Expert Survey and subsequent piloting of the User Survey, it became clear that User and Novice could be conflated without sacrificing data quality.

The other side of the observed phenomena is the editing software itself. This was measured on two occasions, in order to keep pace with the release of new versions. As reported in Flynn (2006), 72 specific actions were identified as available for the control of document structure in a structured editor. By combining these according to their functions, using guidance from the Expert Survey and the Requests Analysis, 12 key functions emerged as representing the most problematic actions when writing or editing structured documents.

The demand for features required in editing software was measured using an analysis of the posts to mailing lists and newsgroups, also reported in Flynn (2006). This provided a list of features which was used to filter the action list from the software analysed.

Finally, a User Survey was made among users of the same online forums used for

the Requests measurement. This gathered data about the way authors and editors performed a defined list of tasks, based on the 12 key functions identified earlier.

1.3.5 Testing

The final stage of the exploration was to identify alternative ways of presenting the 12 functions to the user, and testing them. The 'presentation' could be a changed affordance, or an additional way of accessing it, or a change in the behaviour of an existing affordance, or a combination of all three.

The yardstick of 'suitability for writers' for these changes was that they should:

1. allow the program to achieve the desired result (storing accurate markup of the information);
2. without interfering in the process of writing (any more than the current interfaces do, and preferably less);
3. unless there was a compelling reason evident to the user (for example, better quality, faster response, improved reusability, etc).

As reported earlier (Flynn, 2006), no one editor met these criteria for suitability for writers. A few came close, and there have subsequently been a few systems still under development which have started to implement some of the features we examined.

The 'redesign' of the 12 functions was based on the results of the Requests Analysis and the User Survey, identifying the requirements of the users and the requirements of the markup. In most cases this was self-evident, as there was a clearly-defined and well-known behaviour expected for a given function; in some cases there was ambiguity, and we had to select what appeared to be the least problematic choice.

The method of testing had to be achievable within the constraints of the project: building a complete editor from scratch was not possible, so paper prototyping was chosen for its ability to simulate the exact sets of choices facing the user, while providing a rigorous framework for presenting them, and allowing stage-by-stage recording of the actions.

Chapter 2

Research into the writing and editing of structured documents

1. Usability — Development — User-Centred Design — User experience and Users' Experiences — Affordances — Testing and measuring.
2. Editing software — Historical perspective — Usability and structured editing software — Design processes. 3. Structured documents — Technical writing — Document type design — Typographic design — Interface guidelines — Synchronous typographic editing. 4. Summary.

> *My own experience has been that the tools I need for my trade are paper, tobacco, food, and a little whiskey.* (Faulkner, 1956)

The principal fields of study in this research — usability, software for writing and editing, and structured documents — have each been objects of investigation for many decades, and are extensively documented. While our research is concerned with some specific aspects of interface design and how writers interact with the software, we must also take some account of two closely-related fields: document type design and typographic design. These two areas control, respectively, the nature of the available markup and the visual appearance of a document (section 2.3.2 on page 93 and section 2.3.3 on page 95).

There is a very large but entirely separate field in the psychology of writing *per se*. This contributes much to theories of writers' internal concepts of structure, as well as to their expression with respect to the narrative. As we have seen in 'Literary

view of structure' in the list in section 1.1.3.1 on page 16, it involves a very different notion of 'structure' to ours, although there are small areas of overlap in the technology, such as the use of outliners (section 2.2.1.4 on page 81), and the use of block-move operations which respect the document structure (see, for example, Figure 3.27 on page 196). Flower and Hayes, while developing their cognitive process theory of writing, noted that the writing process itself may be considered hierarchical, especially in respect of the planning and revision process, which may at least have some implications for the way in which writers attain their goals (Flower & Hayes, 1981, Figure 3). Software has been developed to take account of the requirements of didactic writers which allows them to connect their concepts of structure with those used in our sense of the word (Murray, 2003).

Closer to the present subject-matter, Owston et al. (1992) studied the effects of wordprocessing on young students' writing quality and revision strategies. A sample of 111 children from four classes (US 8th Grade: equivalent to the Irish 2nd Year) with 18 months' experience of using computers for English composition was calibrated for mastery of the software (Microsoft *Works*) and then wrote two set essays, one by hand and one by computer. The handwritten work was transcribed for computer so that it was not distinguishable from the computer-written work, and all papers were then assessed by teachers trained and calibrated to rate the writing using one of the Center for the Study of Evaluation's standard scales (Quellmalz, 1982). The scores were analysed using a doubly multivariate repeated measures MANOVA design. The computer-written papers were rated significantly better than the handwritten papers; no significant differences in spelling were found between the two writing media. While the composition of schoolchildren may seem remote from the current study, those children are now (2013) in their mid–30s and may well be very much in our target population. This study is also interesting for its analysis of the use of editing functions, to which we shall return in section 4.3.1.4 on page 240.

Within the study of writing, however, specific domains also have their own body of research and literature: technical writing, business writing, academic writing, 'creative' writing, writing for the Web, writing for children, and countless others. Academic writing has its own unique set of goals (Boice & Jones, 1984) and is very heavily dependent on structured documents. Technical writing is also accorded special consideration here, as it is by its nature the area where advances in our primary fields are first tested (see section 2.3.1 on page 92). Despite the fact that this research concerns itself with writing by authors who are quite

explicitly *not* professional technical writers (see section 1.2 on page 42), the development of technical writing systems themselves remains important because it includes many of the features which we will be investigating.

2.1 Usability

As was mentioned briefly in section 1.3.1 on page 45, the impetus for the study of usability today came from work done in HCI in the 1970s and 1980s. The foundations of HCI itself, however, were laid many decades earlier in work done in human engineering and human-machine interaction in the industrial and military fields since the First World War; and later in the nascent field of 'computing science' (Kirakowski & Corbett, 1990, Chapter 9).

Usability has been defined as 'the quality of a[n interactive computer] system with respect to *ease of learning*, *ease of use*, and *user satisfaction*' (Carroll & Rosson, 2001, p. 9, my emphasis). Kirakowski and Corbett (1990, p. 174) give an example of restricting this in the context of a specific study to 'the speed of learning of users from population *x*'. Ease of learning and speed of learning may not necessarily have a direct relationship: they would depend also on the skills and knowledgeability of the learner, possibly acquired over many years. We will return to this later in section 2.1.3 on page 65.

2.1.1 Development

There is general agreement that Shackel's 1959 paper 'Ergonomics for a computer' and Licklider's 'Man-Computer Symbiosis' (1960) mark the start of human-interface considerations for computing, and the field developed very rapidly thereafter, attaining a much wider coverage than the early ergonomic approaches might have indicated, possibly due to the increasing rate of technological change (Dix, 2010; Diaper & Sanger, 2006; Shackel & Richardson, 1991). Equally importantly, however, these two papers marked a significant move away from the moulding of people to fit machines, to the moulding of machines to fit people.

This paradigm shift is highlighted by Werby (2010), who makes a connection between the post-WWI move from a search for pilots more suited to the aircraft to a search for ways to make the aircraft better suited to pilots, and the post-WWII move from a search for people more suited to be computer programmers to a

search for ways to make the programs better suited to the users; 'reflecting on the effect of the first stages [of engineering usability] on people' (Kirakowski, 2008). A similar move occurred during the 1950s–60s with the post-war spread of consumer domestic devices (televisions, fridges, freezers, etc) and its effect on the redesign of existing devices (radios, telephones, cars). In the fields of industrial design and business management, a closely-related move (from matching people to the device to redesigning devices for people) appears to have occurred a little later (Papanek, 1972; Schumacher, 1973; Pilditch, 1976, all *passim*), although foreshadowed by the various schools of 'new management' thought in the late 1950s and early 60s.

In a summary of the development of the discipline of usability, Carroll and Rosson (2001) point to the pace of Information Technology (IT) development, the spread of usage from IT professional to end user, and the growth of network communications as major contributory factors (pp. 9–14). A later addition places the emergence of usability from cognitive engineering at the point where the Personal Computer (PC) was gaining traction (around 1980), and mainframe software management was starting to consider aspects of development other than simple functionality (Carroll, 2009).

In a note on the early history of the field, Dumas (2007) says that (at the time) it was 'a new area in which to apply traditional methods rather than a one requiring its own unique methods', and explains that it wasn't until new techniques for testing and assessment were developed that usability came into its own (p. 55). This view seems to be supported by Dix (2010) in his description of the efforts to create a discipline of usability, and of the hotly-debated arguments for and against — a debate paralleled in another field more recently, in the effort to gain recognition for humanities computing as a discipline (McCarthy, 1999; Burnard, 1999; Orlandi, 2002): one from which some threads of this present research evolved.

A second paradigm shift occurred in the 1980s, when greater emphasis began to be laid more on the cognitive aspects than the physical ones. In the presentation of this approach, the individual becomes analogous to a computer, with inputs, storage, a processor, and outputs (Card et al., 1986). The advantage to this stance from a technical viewpoint is that the chain of events (signals and actions) can be 'pipelined' from the external environment, through the subject, into the computer, and back out into the real world, providing a homogeneous framework for analyses and comparisons of human-computer interactions. This generalisable

model (curiously paralleled in other fields unrelated at the time, such as markup theory) enabled the development of a number of concrete models for the analysis of specific cognitive applications, such as GOMS (Card et al., 1986, Chapter 5). The combination of the Human Factors and cognitive approaches is credited with contributing largely to the emergence of many aspects of HCI that are now commonplace, including the GUI (Harper, Rodden, Rogers, & Sellen, 2008).

2.1.2 User-Centred Design

A third and more recent shift has been the introduction of User-Centred Design (UCD), although its roots are in the cooperative approach in industrial design adopted from the 1970s onwards, as referred to in section 2.1.1 on page 57, which involved the participation of designer and users working together.

The case for participatory design was strengthened by the first international standard for usability, ISO 13407:1999: 'Human-centred design processes for interactive systems' (ISO TC 159/SC 4, 1999), which aimed to make the design process more holistic, multidisciplinary, and user-driven, bringing together a number of aspects including basing the design on known user requirements and testing the result recursively with users at each stage (Bevan, 2001), a theme which we discuss in section 2.1.5 on page 70.

Gould and Lewis (1985) presented three key principles in design:[1] *a*) an early focus on users and tasks; *b*) empirical measurement; and *c*) iterative design. They tested a large group of systems designers and programmers at a human factors presentation to see to what extent they would mention the principles when describing the steps they would take when designing a system. While 62% mentioned an early focus on users, only 40% mentioned empirical measurement, and only 20% iterative design. In an extensive discussion, they use a case study to demonstrate that using the three principles can improve usability. Ultimately however, they say:

> It is our experience that people sometimes lack the ability to differentiate between what we recommend and what they do (p.301). [...] Survey data show that these principles... are not intuitive (p.311).

[1] The original paper (Gould & Lewis, 1983) describes four principles: *a*) identify the users; *b*) user panel working with the designers; *c*) measure performance and reactions early; and *d*) use iterative design and fix the problems. The 1985 article mentions these four, but refines them to the three shown here.

Further work by Gould (1987) reverted to four principles (early focus, empirical measurement, iterative design, and integrated design) and produced a useful prescriptive design methodology. Hewett and Meadow (1986) reported success using this methodology in a proof-of-concept case study of an helper application for bibliographic searching.

The phrase 'user-centered design' gained more widespread attention with *The Design of Everyday Things* (Norman, 1986), bearing upon the theme of suitability to the user's immediate needs, a theme which Norman later refined in *Emotional Design* (2004). The methodologies based around this have been enthusiastically adopted in many areas, and have contributed to a revision of the standard as ISO 9241-210:2010: 'Ergonomics of human-system interaction — Part 210: Human-centred design for interactive systems' (ISO TC 159/SC 4, 2010).

The essential component of UCD is the emphasis on the human experience rather than on the technology. Apple's *Apple Human Interface Guidelines* (Apple Corporation, 2008), heavily promulgated by its authors (Tognazzini, 1992), have contributed much to the idea of [re]writing the interface so that it needs less explanation, rather than spending more on training users in what are perceived as more hostile interfaces. This tendency is evident in the more recent development of 'Web 2.0' interfaces, which can require virtually no training or prior exposure (section 2.1.2.1 on the facing page); 'design as an art-form' (Kirakowski, 2008).

Anecdotal and empirical evidence (by no means all of it in the field of usability) is adduced by Landauer (1995), Tognazzini (1992), and others of significant improvements in usability when concentration on technological superiority is replaced by concentration on satisfying the user's needs. By extension, improved usability should lead to more tangible benefits such as greater productivity and user satisfaction, reduced frustration, lower error rates, and less need for support; and, in the commercial field, higher sales. It is perhaps unfortunate that some of the evangelism for UCD in IT and elsewhere is demonstrated by proving the negative: that failure to use UCD is responsible for failure to achieve these goals (Cooper, 1999; Underhill, 1999).

User-centered design does not imply that the technology should be ignored; rather, that it should be deployed to meet user requirements instead of taking precedence over them. UCD shifts the balance so that the technology does not overshadow or obscure the inherent requirement to create products that fit the human, rather than force the human to adapt to the product (the same process as alluded to earlier in section 2.1.1 on page 57.

The involvement of the users, or representatives of the perceived or target user community, is key to the process of UCD. This provides a basis for the contextualisation of the process, so that products can be evaluated and re-evaluated in the same circumstances that they will be used. The cyclical nature of iterative testing also implies communication, in both directions. In the temporary absence of the users who are testing the product, (for example, between tests) scenarios and personas can be used to help in framing the tests.

2.1.2.1 The Web and UCD

While the genesis and development of UCD took place while the web was non-existent or in its first few years, the field of website usability has become the primary focus for many practitioners. Because of the web's ubiquity, and because it has become a platform on which other products are built, it is often the first place that an IT product will appear. Nielsen (2000) has noted that on the web, users can experience the product before they buy, whereas with traditional IT products, they buy first and experience afterwards.

This is perhaps less true since the rise of Free, *Libre*, and Open-Source Software (FLOSS), which provides the software product for download and typically requires no purchase. It is even more true of cloud computing, where the software resides remotely, and only a (relatively) small interface driver executes locally (in the browser). Nevertheless, availability before acquisition still has a significant impact on the user experience of the product as well as being itself impacted in turn by the user's cumulative experiences.

Historically, developments in computing used interfaces designed and programmed by the developers themselves, based directly on the underlying operating system (Kirakowski, 1988, Chapter 3). These developers were skilled programmers working with one class of user in mind: other similarly-skilled programmers. The users were presumed to be capable of the same level of knowledge, and of making the same sets of assumptions as the original developers, and therefore required minimal documentation, a command-line interface or API, and only the tersest of error messages (Cooper, 2004, p. 22). However, at some point in the maturity of a program, the more recent FLOSS development methodology of collaborative work takes over, with dozens, even hundreds, of programmers contributing bug fixes and other developments (Feller & Fitzgerald, 2001, Chapter 6).

The spread of the Internet, the web, and FLOSS development has meant that many of these paradigms still remain in place, and have been quoted, misquoted, and parodied, in many places (Raymond, 2004).[2] It should be noted that the Internet itself is formally viewed as a 'network of networks', and that at the administrative level, 'user' means 'systems manager responsible for the connection of a network to the Internet', not an end-user at all.[3]

However, first-generation web user interfaces (browsers, as distinct from interfaces to other Internet methods such as File Transfer Protocol (FTP) or email clients) were heavily conditioned by the requirements at the time (communication between particle physicists, not computer experts). The inventor also had a sophisticated development tool (a *NeXT* workstation with a graphical interface), and the other early graphical developers (of *Mosaic*, *Netscape*, etc) were also users of the Unix-based X Window System. Second-generation web browsers were therefore perhaps the first really large class of interfaces designed explicitly with the completely non-IT-literate end-user in mind.

The considerable amount of usability evangelism (Tognazzini, 1992; Johnson, 2000; Cooper, 2004; Krug, 2006) directed specifically at web-browser interfaces has demonstrably affected the interface design process — the 'Web 2.0' paradigm mentioned earlier is one good example; the interfaces used on mobile-phone and tablet applications are another.

2.1.2.2 Intuitiveness and obviousness in UCD

A key word in user discussions of interface development is 'intuitive'.[4] Formal (dictionary) definitions describe it as the ability to 'know' something *without foreknowledge* (my emphasis), without deducing or inferring it or thinking it through. However, this would appear to be at variance with the implications of

[2] Common examples being "If you can't understand it, you shouldn't be trying to use it"; "if you don't like it, or you find a bug, you can always fix it yourself"; and 'if it was hard to write, it should be hard to use.'

[3] This author once had to write a report as the European Observer to the 1990 National Science Foundation Network (NSFNET) meeting which helped establish the US Interim National Research and Education Network (iNREN), in which it was necessary to explain the European usage of 'user' to mean 'end-user', as it was virtually unknown with that meaning in [US] Internet circles at the time.

[4] As has been pointed out many times — for example, Tognazzini (1992, p. 246) — the sense is reversed: it should probably be 'intuitable', or even, as Tog further suggests, 'articulate', given that the interface 'speaks to' the user.

the active voice of the back-formation 'intuit', which implies some degree of cognition.

Intuition is popularly regarded as a form of near-immediate grasp of the utility of an affordance (J. J. Gibson, 1977, pp. 130–136), which we discuss in section 2.1.4 on page 66. Although speed of *perception* is not discussed in this context, as it is biologically close to instantaneous, speed of *cognition* is implied in the concept of near-immediacy, which in turn implies that it is in some way dependent on recall of the user's prior experiences, and does not require extensive cogitation. Gibson does indeed discuss this (J. J. Gibson, 1977, pp. 253–254), but only in terms of perception, and then only in relation to his theory of Information Pickup (see section 2.1.4 on page 67). Marr (1982, p. 29) correctly points out the complexity of computational ability required for processing the perceptual information on physical invariants that Gibson uses in his examples, but this is at a level of analysis several orders of magnitude finer than experienced in an editing interface. Marr has in turn been taken to task (*en passant*) for himself underestimating the complexity of the process (Burke, 1998, p. 89).

Discussion of intuitiveness in web interface circles has been much to the fore since the rise of the 'Web 2.0' interface paradigms referred to above, especially with regard to their use by younger users with no foreknowledge of more traditional interfaces. Much of the discussion centres on the expectations of the users — for example that everything is clickable; that everything can be edited *in situ*; and that there is an affordance for everything you might want to do with the information (what one colleague has described as 'tweet it, repeat it, delete it') — and by corollary, if there is no perceivable affordance, it can't be done. We may speculate that the cumulative effect of exposure to such rich interfaces, with the only examples (patterns) needed being provided by an almost trivial handful of guidance moments from older siblings at an early stage, has both raised the expectations of young users and provided them with extensive experience of what you can and cannot expect to be able to click on (see section 2.1.3 on page 65).

In older users, however, this may or may not be the case, depending on their level of exposure, their knowledge or understanding of both new and traditional interfaces, and the extent to which even older and more experienced mentors were available.

We will be using the term 'affordance' sparingly. The example of the Print button mentioned in section 2.1.4 on page 66 below is useful here: to anyone with even a small exposure to computing there is a clear affordance to get something

printed, and the button design is familiar to everyone with exposure to virtually any form of machinery, including the ubiquitous mobile phone.[5] Identifying the function of a button labelled with a well-known feature is certainly a mental process, involving prior knowledge, and the affordance of such a button would be referred to popularly as 'intuitive', whether or not you subscribe to Gibson's or Norman's view of the precondition of perception, which we examine in section 2.1.4 on page 66 below.

Obviousness is a phenomenon closely-related to affordance, and it could be argued that it forms the basis on which both affordance and intuitiveness are based. Some interfaces take this to extremes, with [OK] buttons that are extra large and brightly coloured, possibly because they were being overlooked by users when they were the standard size; but equally possibly because they represent a programmer's perception of the users' [low] level of skill, or because exceptional conditions such as safety in life-critical circumstances require more than ordinary caution to be used. Obviousness is also closely related to the 'principle of least effort' in linguistic analysis (Papanek, 1972, p. 186), a point which has not been lost on the author of at least one guide, *Don't make me think: A common sense approach to web usability* (Krug, 2006).

Given the global nature of IT, and particularly the editing interfaces, there is also clearly scope for very serious errors to be committed when imputing obviousness across linguistic or cultural boundaries: the choice of words or images may simply be wrong or inappropriate. But leaving those factors aside, there remain the other considerations such as location, size, colour, and naming; or there may be a position away from which an interface item (widget, control, display) might be said to be insufficiently obvious.

2.1.3 User experience and Users' Experiences

'User Experience' (UX) describes how a user feels about using a product or service. It relates to the cognitive-affective aspects of a user's interaction with the target,

[5] It has been traditional at this stage to add that much of this level of knowledge is restricted to the first and second worlds, but the recent penetration of the mobile phone, as well as other devices, now means that exposure to the idea of pressing a button may be approaching universal. The recent experience of the One Laptop Per Child (OLPC) team in blind-dropping instructionless tablets into remote and unconnected Ethiopian villages (Negroponte, 2012), while so far unvalidated and unreplicated, offers an interesting insight into the deductive, creative, and experimental capabilities even of children with no prior exposure to IT.

and to the user's perception of its features, such as usability (ISO 9241-210:2010, 2010). Many factors contribute to this, and part of many UX design methodologies is given over to measurement and analysis of them. The term was in use in the early 1990s (Tognazzini, 1992, pp 26,71), and (in the form 'interface experience') even earlier (Norman & Draper, 1986). It was mentioned almost *en passant* in a presentation abstract by Norman, Miller, and Henderson (1995), which is frequently cited as the origin, but has since become the conventional descriptor of the way a user feels about a product, service, or even a company, and is in widespread use well outside the field of usability.

However, in the present research, we are not attempting specifically to design for such a wide scope. While the UX of an editor in terms of design is clearly of great importance, we will be relying on another facet of users' interaction with systems: their cumulative experiences to date of related products and of computer systems in general, as we referred to briefly in section 2.1.2.2 on page 62. Finding a term to describe this which is sufficiently distinct from UX has proved challenging, but we have been led by the phrase **users' experiences** (Laurel & Mountford, 1990; Flynn, 1997), which we may designate UUXX.

There has been some understandable reluctance to admit this facet to the canon of studies of usability. It involves breaking the hermetic seal and including elements of users' past interactions with related areas of involvement that broaden the scope of an inquiry beyond reasonable bounds. Because of the difficulty of gathering data on users' historic experiences, they (the experiences) are usually excluded from any analysis, although they are certainly mentioned (Owston et al., 1992). However, the individual user's ability to understand a system depends partly on their accumulated knowledge and experience (Maass, 1983, p. 25), and for this reason we will include questions relating to this in some of the inquiries in section 3.4 on page 176.

By way of example, at one end of the continuum of experiences, a user's past involvement with computers may extend back to the days of mainframes, and to editing plaintext *RUNOFF* files with *TECO*. At the other end, a user may have experienced only *Word* and *Facebook*, and the only experience of markup have been clicking on the I button.

2.1.4 Affordances

The theory of affordances was introduced by J. J. Gibson (1977, 1979, Chapter 8) but has been developed and extended over the years (Gaver, 1991; Norman, 1988; Sieckenius de Souza, Prates, & Carey, 2000) to include or exclude certain aspects of perception or resolution.

The concept of an affordance as originally introduced is explained quite straightforwardly (with reference to the 'ecology', or physical environment, which J. J. Gibson (1979) is discussing) as 'what it *offers* [...], what it *provides* or *furnishes*' (his emphasis, p. 127). He explicitly derives the concept from the Gestalt psychology of the 1930s and the *Aufforderungscharakter* proposed even earlier by Lewin (1926), but he makes it clear that in his (J. J. Gibson's) terms, 'affordance' is an absolute phenomenon, and does not rely on the perception of the observer (J. J. Gibson, 1979, p. 139). This is necessary to overcome any dependency on the observer's needs or foreknowledge (or lack of it), otherwise water would only say 'Drink me' when the observer is thirsty (*pace* Koffka). Lewin's infant grasping for a shiny toy is attracted to its surface, without necessarily knowing what it is *for* (Lewin also refers to the concept as *Reiz* or 'attractiveness').

This independence of any observer derives from the key aspects of Gibson's considerations: in his terms, an affordance is an abstract phenomenon of the the invariant properties of light specific to the reflecting environment, registered in the optic array. At this physical level, Gibson highlights the *potential* visibility of a surface (J. J. Gibson, 1979, p. 23) and the *potential* stimulation at a point (p. 53), and acknowledges the need for perception to occur before an affordance can be *taken advantage of* (pp. 36–37, my emphasis): he is quite explicit about the absence of perception:

> The observer may or may not perceive or attend to the affordance, according to his needs, but the affordance, being invariant, is always there to be perceived. An affordance is not bestowed upon an object by a need of an observer and his act of perceiving it. The object offers what it does because it is what it is. (J. J. Gibson, 1979, p. 139)

By isolating the existence of an affordance from the need (or otherwise) to act upon it, J. J. Gibson freed himself to develop his theory of Information Pickup, in which he defined 'perceiving' as a combination of persistence (invariants) and disturbance (changes) in both the environment and the beholder (J. J. Gibson,

1979, p. 249). In our present research, what is revealing about his discussion is his comments on repeated exposure and learning:

> If this recurrent sequence [event B following event A] is experienced again and again, the observer will begin to anticipate, or have faith, or learn by induction, but *that is the best he can do*.
>
> [...] According to pickup theory, information does not have to be stored in memory because it is always available. (J. J. Gibson, 1979, p. 250, my emphasis)

We shall be examining the role of learned or remembered behaviour as it applies to interfaces in section 2.1.3 on page 65.

Greater awareness of affordances in everyday objects as well as in computer interfaces is largely due to Norman (1988), who identifies hundreds of examples of misleading or inadequate affordances. The affordances of an object are defined in this context as the 'fundamental properties that determine just how the thing *could possibly be used*' (p. 9, my emphasis). Nevertheless, for Norman, these fundamental properties must be perceived in order for the affordance to exist, a stance which conflicts with J. J. Gibson's view, and which was later clarified in a more emphatic paper (Norman, 1999). However, J. J. Gibson is quite clear that an affordance is an objective property of the visual array, and Norman's view is essentially untenable outside his narrow view of the perceptions of the user. In our treatment of UUXX, we follow J. J. Gibson's definition.

A more nuanced approach is taken by Gaver (1991). In his view, affordances are "properties of the world that are compatible with and relevant for people's interactions" (Gaver, 1991, p. 79), and he distinguishes principally three classes of affordance (shown here with their equivalents from J. J. Gibson's standpoint in vision):

Gaver	**Gibson**
False (looks like one, but isn't)	Visual illusion
Hidden (it is one, but it's not evident)	Beyond optical resolution
Perceptible (Norman's definition).	Invariant pickup

This distinction is especially useful when we consider computer interfaces, because it allows us to take account of users' acquisition of cognitive skills over time (our UUXX), which may allow them to identify an affordance as such or reject it — Gaver later gives this as a fourth class, Correct Rejection, which equates to perceptual learning in Gibson. He sums it up as 'perceptual information may

suggest affordances that do not actually exist; while those that do may not be perceivable' (Gaver, 1991, p. 80). While Gibson's 'attunement to invariants' underlies his theory of affordances, it is a progressively greater attunement to higher-order invariants that supplants the idea of memory as a filing cabinet.

Norman's proviso that an affordance has to be perceived as such, if it is to be of any use, is fortunately in accordance with the precepts of user-centred design, although more by accident than design; and it has also been argued persuasively that this all-or-nothing approach interferes with the analysis of how the user makes sense of the interface (Sieckenius de Souza et al., 2000). In introducing the parallel concepts of 'missing' and 'declining' affordances, these authors extend Gaver's analysis into a useful tool for measuring the *degree* to which an object in the interface may be perceived as an affordance. Classing some affordances as missed or declined (the preterite is easier to express than the authors' adjectives) allows us to judge their success or failure in a manner which the absolutist approach denies.

Affordances (in Norman's terms or otherwise) are a key factor in studying editing interfaces to structured text because many of the interfaces differ significantly in name, appearance, location, or operation to their wordprocessor counterparts, even though their ultimate (formatted) result may look the same.[6] It was the wishful thinking by interface designers and program authors who were claiming objects in the interface as affordances that caused Norman to clarify his position that he had been referring to perceived affordances, not 'real ones' (Norman, 1999, p. 39). However, we continue to argue that there is a third factor to take into account, beyond perception or non-perception, and that is a composite of the users' 'skill', knowledge — or lack of it — and their prior experiences (the UUXX which we mentioned earlier).

Affordances (in our current circumstances) are objects in the interface which invite a purposeful action by the user, and provide the capacity to influence the working of the system. A button or toolbar or menu item in a conventionally visible position which shows a printer icon or says Print (in the user's language) and causes the document to be printed, in our view *unequivocally* affords the user the opportunity to print the current document (Gibson's terms). If the user is

[6] A case in point is the distinction between 'new line' and 'new paragraph', which will be familiar to anyone who has worked with text editing (see Figure 4.3 on page 231 and section 4.5.4.3 on page 287), and revealed in the analysis of our Test 3 (section 5.2.3 on page 310); as is the distinction between the different reasons for the use of italics (see the list on page 36).

unable (or otherwise fails) to recognise the word 'Print', or perhaps fails to recognises the icon as a printer, perception does not occur (Gaver's Hidden affordance: there is no pickup of the information). The problem with Norman's assertion that failure to perceive renders the affordance null and void is that it fails to take into account those occasions when the programmer or interface designer has done everything reasonable, and in some cases gone well beyond that, to make the item obvious (perceivable), but the user has either not seen it or has failed to understand it: a failure of pickup which we submit may be due to the user's expectations — a result of their UUXX — interfering with the exposure of the affordance.[7]

It is argued that in many circumstances a user's failure to perceive or pick up an affordance is a failure of the interface and its designer, not the user (Cooper, 2004). It is indeed the designer's task to *a*) provide the affordances; and *b*) ensure that the user for whom they are designing can pick them up unambiguously, but we would take the view that beyond certain points the interface [designer] cannot reasonably be held to account. While in the natural world there is arguably no-one to correct someone's failure to perceive an affordance, in the artificially-created world of computer interfaces there are (one hopes) designers taking note of the failures in order to correct them. The fundamental problem with Norman's approach is that it restricts the application of the term 'affordance' to those which the designers have designed specifically because they want them perceived solely for the purpose for which they intended them, and not otherwise.

(It is worth noting that the use of standardised generic markup in plaintext files, whether XML or LaTeX, enables multiple authors on the same document to use different software (read 'set of affordances' or 'pickups') that suits their level of skill and way of working, without affecting the resulting document markup.[8],

[7]User expectation has received relatively little attention in recent research into interfaces. The prevailing assumption that user *experience* is the sole conditioning human factor of usability neglects the circumstances of new users, where experience is by definition zero, but there is still a set of expectations. We have argued above that these are the result of user experience in other, related domains, but as this is hard to measure, admitting it to consideration risks polluting the value of user experience measurement in other areas. In western domestic and educational culture, where the use of computers is pervasive, and their misrepresentation widespread, especially in the mass media, it can be argued that virtually everyone has acquired their own set of expectations of computer systems *whether or not they have ever used a computer*, which may in some cases may be far beyond their actual capabilities.

[8]This is less important in some areas than others, however: Larivière, Gingras, and Archambault (2006) point out that the vast majority of papers in the Humanities are

which implies that there can indeed be many different affordances for any given result.)

It seems, therefore, to follow that there are circumstances when it is not the vendor's, programmer's, or interface designer's 'fault', as they could be said to have done their best to make the affordance perceivable. We are aware that this is thin ice here, as the inherent supposition of UCD, mentioned earlier, is that the user is always right, and that failures of the interface are always due to a design error. However, this approach fails to take account of the very wide range of variability among users' abilities and experiences. We said above that the designer might not be held to blame 'beyond certain points', and the plural was deliberate. An editing program, whether a simple plaintext editor, a wordprocessor, or a structured-document system, may be used by anyone from an expert to a novice, outside the control of the vendor, and the skill endpoints would form the tails of a distribution of skills. It is unfortunately outside the scope of this research to investigate the distribution of skills, as we are concerned with a specific, narrower set of users well inside the bell-curve, but the variability of the target population perhaps forms another argument for the inclusion of UUXX in test data collected.

2.1.5 Testing and measuring

Techniques specifically for testing and measuring usability derived largely from the fields of cognitive and applied psychology, and from management science, where 'user satisfaction' is a well-established concept. However, the pressures of commercial product development mean there are far more **iterative tests** (many in the course of development) than **acceptance tests** — typically, one at the end (Lewis, 2012). The demands of iterative testing, combined with the implementation of changes at each stage to prevent tester frustration, led to the need for more frequent testing, and for testing earlier in the development cycle, although it is arguable that the tests are still not frequent or early enough.

The many methods and techniques of usability testing (Wildman, 1995) fall principally into two groups: those tests conducted with users, and those conducted without them.

User present: In classical user testing,[9] a sample of users or potential users is

single-authored, whereas almost all papers in the natural sciences are multi-authored (Larivière et al., 2006, p. 524, Canadian sample).

[9] 'User testing' is an unfortunately misleading phrase: it is normally the product which is being

asked to complete a set of predefined tasks, either alone (**unmoderated tests**) or under observation (**moderated test**) — Barnum's contention that unmoderated testing means web-based testing (Barnum, 2010, p. 41) is surely inapplicable to some forms of test.

In **questionnaire-based tests**, results are measured against standardised and calibrated baselines. This method can produce very accurate measures of user responses such as satisfaction (Kirakowski & Corbett, 1993): it is of more limited use for fine detail.

Performance-based tests measure data from behavioural observation, captured by video or by modifying the software being examined to log the data, such as time-to-completion or error rates per completed task (a measure of effectiveness). Wildman (1995) notes that this is can be a costly approach, given the equipment or preparation required, and the volume of data generated.

Cognitive modelling derived from earlier work on cognitive engineering (Card et al., 1986; Olson & Olson, 1990), notably the GOMS family, and especially Keystroke-Level Modelling (KLM). It relies on finer disaggregation of the tasks than most other methods, but is useful for assessing against narrowly-defined goals (Mayhew, 2005, p. 466).

User not present: Expert **inspection tests** measure a product heuristically (Barnum, 2010, p. 61): it is evaluated point by point against a known set of criteria obtained from prior observation (in a similar manner to the questionnaire method above) or from expert or common knowledge Nielsen and Molich (1990); Nielsen (1994).

Structured walkthroughs compare goals with tasks, and attempt to identify areas where the user would be expected to have problems; **cognitive walkthroughs** evaluate a product for ease of learning (Wharton, Rieman, Lewis, & Polson, 1994). Like other inspection tests, walkthroughs rely heavily on the depth of knowledge of the experts involved.

Iterative tests go beyond the presence or absence of users, and are used to provide ongoing feedback for error correction or product modification (**formative tests**); acceptance tests are done on completion or at stages, against some known benchmark, and are used for overall evaluation (**summative tests**).

---

tested, not the user.

Formative methods were distinguished from summative methods in the educational curriculum field by Tyler, Gagné, and Scriven (1967), based on a need to identify different types of intent (to measure with feedback during development, as distinct from measuring results against goals). Early use of formative tests in computer usability was described by Al-Awar, Chapanis, and Ford (1981), and it now forms a standard part of usability testing (Rubin, 1994).

Summative testing appears not to have been applied to software until the 1980s but was codified in an ESPRIT project 'Measuring the Usability of Systems in Context' (MUSIC) which was informing ISO 9241-11:1998 (Kirakowski & Bevan, 1997). Much later, Bevan (2006) has also emphasised the focus on the end-result for summative tests, including the need to distinguish partial goal achievement, relative user efficiency, and productivity.

While in both cases, measures are recorded for the data required (timing, accuracy, completeness, errors, etc). Wildman (1995, p. 22) offers some concerns about the applicability of numeric methods to 'the big picture' instead of the 'discrete operations' (with which we are concerned here). In an attempt to point the way forward, McNamara and Kirakowski (2006) tied the design and evaluation of technology — specifically in respect of HCI — to Functionality (features, performance, reliability, durability); Usability (transparency, learnability, support; but not 'intuitiveness'); and User Experience (on which they pass the buck to some extent on 'design reductionism'). This approach provides a good framework for designing evaluations, and in giving due weight to Functionality it avoids the charge of placing the 'infinitely variable' measurement of the cognitive concerns of usability and UX ahead of the more binary measurement of function ('does it work or not?').

Kirakowski and Murphy (2009) considered a number of metrics in a study conducted at a usability conference. Despite the participants being from organisations involved in usability, the results were disappointing: the authors identified 'major conceptual flaws in terms of measurement'. While acknowledging that the wide spread of data was unavoidable, they recommended gathering both quantitative and qualitative data, developing a more standardised set of measures (especially Time-on-Task), and the use of WAMMI. Remote unattended testing (which is considered for this study in section 1.3.2 on page 45) was deprecated for lack of precision.

2.2 Editing software

Several different but closely-related threads need to be considered here, as the evolution of the software (editors and the formatters which process their files) is closely bound up with the evolution of the document model and the markup used to realise it.

The timeline in Figure 2.1 shows the parallel evolution of prevailing markup paradigms or standards (top band, in green), some of the significant editing software (middle band, in blue), and some of the related processing software (bottom band, in pink). The invention of the World-Wide Web (WWW) in 1989–90 was arguably the biggest paradigm shift, as HTML brought the concept of markup into the realm of public awareness for the first time.

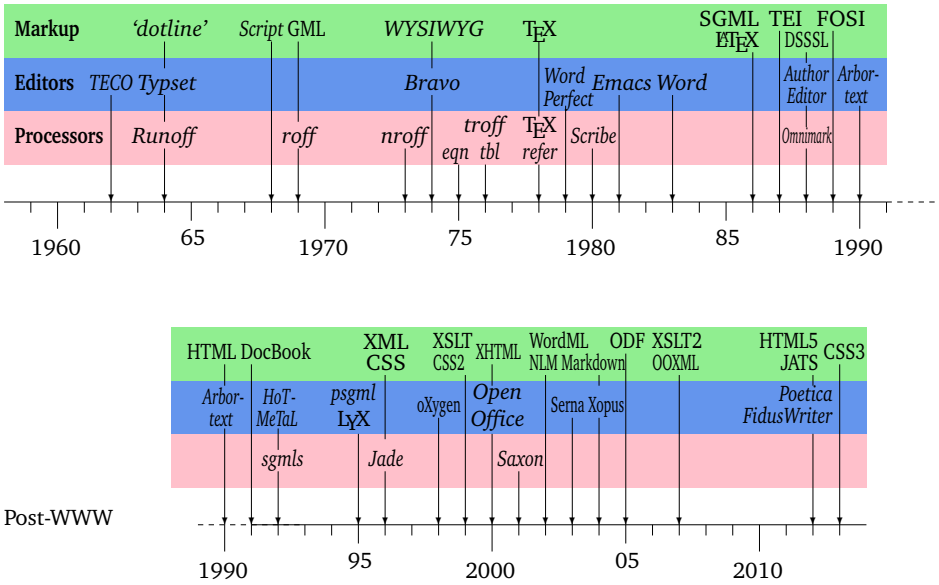


Figure 2.1: Some milestones in the evolution of text editing, processing, and markup

2.2.1 Historical perspective

The earliest editing software was written for programmers to edit programs, and concentrated on speed, power, minimisation of resources, and accuracy, rather than what we would now think of as usability, although programs such as *TECO* would be come to be regarded as perfectly usable by their intended audience: other programmers (Murphy, 2009).[10]

[10]When old flame-wars are resurrected between traditional 'enemies' such as *Emacs* and *vi*, comments on 'usability' clearly refer to the different camps' perceptions of what an editor *ought* to

Within the printing and publishing industry, text-formatting and typesetting software was highly specialist and required dedicated hardware and software (André, Furuta, & Quint, 1989, p. 2). Berg's compendious study of editing and typesetting systems compared most of the common systems of the time (Berg, 1975); while the results are interesting historically, and the methodology sound, very few of the systems described exist in anything like the same form today. The objective of these systems was to set type, not to identify or record document structure. The document model, such as it was, was therefore largely flat and unstructured. A document instance was a stream of characters, some styled differently to the rest, but otherwise undifferentiated.

However, outside the printing industry, particularly in large corporations and in universities, formatting programs were being developed which used a different model. Like the commercial systems, these used one program (the editor) for editing the document, embedding formatting instructions in along with the text; and another program (the formatter or processor) which operated on the saved document to drive the typesetter. But unlike most commercial systems, these programs were starting to use codes which represented the component parts of the document according to their function, not their appearance.

It had of course long been known that certain document components reoccur throughout a document, using the same format or layout each time with different text, as we saw from Vesalius (p 8); much in the same way that fragments of programming frequently reoccur, performing the same operations but with different data. The invention of **macros** (reusable programming fragments) allowed individual formatting codes for an operation to be grouped together and named, a concept which had been around since the development of assembler languages and autocode in the 1950s. While this also happened in industrial systems, commercial pressures on proprietary secrecy meant that information about methods did not circulate in the same way as it did in the research field.

---

do, and how it should do it, compared with how the speaker's own favourite editor does it (Usenet newsgroups `alt.religion.emacs`, `comp.emacs.advocacy`, etc *ad nauseam*), before descending into furious discussions about features. As we have seen in section 2.1.2.1 on page 62, this persists to the present, notwithstanding the comment that '[t]he comparison of widely varying text editors has only recently evolved beyond subjective preference and name-calling' (Borenstein, 1985).

2.2.1.1 Dot-line markup and the use of the macro

Probably the earliest and most influential of the formatters to benefit from this move was *RUNOFF* (Saltzer, 1964). Text was edited with an editor (called *TYPSET*) to add formatting instructions starting with a dot (period) and a keyword at the start of the line — anything else on the line was regarded as a value to act upon, or text to typeset. Thus a level–1 heading would be typed as:

```
.h1 Research into the writing and editing of structured documents
```

The *RUNOFF* processor would read this and output it as a numbered heading, with word-wrapping, capitalisation, blank lines above and below, and other formatting features which had been programmed into the `.h1` macro.[11] Derivatives of *RUNOFF* (such as *ROFF* and *NROFF*) were also developed, including those to support phototypesetters, most notably *TROFF* (Ossanna & Kernighan, 1976). While *RUNOFF* provided commands and parameters for spacing and alignment, the support required for phototypesetting meant that the derivatives included more detailed commands for setting point sizes, fonts, indents, tabs, and other features (Ossanna & Kernighan, 1976; Kernighan, 1978), including the extensive use of the backslash to 'escape' from the current typesetting mode and invoke some special function like a non-keyboard character.

Although *RUNOFF*, *TROFF*, and others were still based around two-character mnemonics for the operations they performed, extensive collections of macros were written for different types of document, foreshadowing the document types and document classes of later systems. In the process, this created the opportunity for naming (or at least, abbreviating) the macros according to the logical function they performed (such as `.h1` for a heading), rather than the physical operations needed to achieve it.

This period is perhaps the *locus classicus* of the distinction between **procedural markup** (detailing the individual physical steps to achieve formatting) and **logical markup** (naming the macro according to the function it performs in the document). The *TROFF* manual has a good example reproduced in Figure 2.2 on the following page.

[11] In fact the original program was not quite as extensive as this: it implemented only one or two styles for headings and subheadings, but it did include possibly the first ever justification algorithm.

> The housekeeping associated with starting a new paragraph should be collected in a paragraph macro that, for example, does the desired pre-paragraph spacing, forces the correct font, size, baseline spacing, and indent, checks that enough space remains for more than one line, and requests a temporary indent.
>
> ```
> .de pg          \"paragraph
> .br             \"break
> .ft R           \"force font,
> .ps 10          \"size,
> .vs 12p         \"spacing,
> .in 0           \"and indent
> .sp 0.4         \"prespace
> .ne 1+\\n(.Vu \"want more than 1 line
> .ti 0.2i        \"temp indent
> ```
>
> (Ossanna & Kernighan, 1976, p. 32)

Figure 2.2: Example of procedural markup (*TROFF* manual, 1976)

2.2.1.2 Generic markup

Work on typesetting systems at IBM for publications, documentation, and legal papers led to two important developments based on the foregoing principles. One was the *SCRIPT* document preparation and typesetting system (Madnick & Moulton, 1968), which inherited the 'dot-line' syntax of *RUNOFF* as well as maintaining the separation between editor and processor; the other was the Generalized Markup Language (GML).

GML attempted to solve the problem of different projects using different procedural markup by creating reusable ('generic') tags for the principal components of a document (headings, paragraphs, lists, etc). It also introduced the concept of **containment**, where one pair of tags enclosed others: for example `.ol` and `.eol` stood for the start and end of an ordered (numbered) list respectively, and between them would be lines starting `.li` for each list item.[12] In 1975, GML was added to *SCRIPT*, by that time officially called the Document Composition Facility (DCF), and the concepts on which they were based were to underpin the work, started in 1978, which eventually became SGML (Goldfarb, 1996).

At this stage, it was still conventional for each application to have its own editor. As editing software became more prevalent, it was clear that any text editor capable of editing plaintext documents could be used to add markup, and the

[12]The reader with some familiarity with HTML will immediately recognise its origins.

need for a specially-developed editor per application ceased to hold much attraction. It was assumed in any case that authors would have no serious difficulty in remembering and typing a dot and a small number of two-letter abbreviations (`.ti` for title, `.au` for author, `.h1` to `.h6` for heading levels, etc). The development of *TECO* as early as 1962 had shown that the **macro** was well suited to use in editors (Murphy, 2009), obviating the need to reinvent an entire editor for each application.

2.2.1.3 Variant syntaxes and extended capabilities

In the early 1970s, several attempts were made to extend the editor/formatter model to a complete **document preparation system**, in the manner of *SCRIPT*. A number of systems also dealt with particularly difficult parts of the problem.

At the time, character sets, non-alphabetic characters, and symbols, especially in technical fields, were not standardised; the fundamental standard American Standard Code for Information Interchange (ASCII) provided only for A–Z, a–z, 0–9, and simple punctuation. Hershey (1972) tackled this with an application designed for scientific rather than general typesetting, and was restricted by the limited device capabilities of the time, but the fact that it addressed a thorny question head-on had some influence on subsequent systems, including *eqn* and TEX.

Much of the development work on text processing in this period took place on smaller 'lab' machines such as the DEC PDP series and Unix systems, although there was a significant level of experimentation with other platforms, especially in universities. The Stanford Artificial Intelligence Laboratory used many platforms and languages, among them the Stanford Artificial Intelligence Language (SAIL), which was used for *PUB*, a combination markup language and scripting language, which influenced both *Scribe* and TEX as well as SGML (Tesler, 1972).[13]

The Unix operating system, which had been developed in 1969, was gaining traction in research laboratories by the late 1970s, and *roff*, *nroff*, and *troff* were becoming common tools for writing research papers, lab reports, technical documentation, and theses. Syntaxes were developed for representing mathematics (*eqn*), tabular material (*tbl*), and bibliographic references (*refer*), usually as preprocessors or macros for the **roff* family (Kernighan & Cherry,

[13] In his annotated manual, Tesler compares *PUB* to today's HTML and PHP.

1975; Lesk, 1976, 1978). These systems also influenced subsequent developments.

In the late 1970s two additional syntaxes joined the dot-line. While the substitution of one escape character for another (whether the dot, the 'at' sign, the backslash, or something else) is not in itself of significance, these two systems (*Scribe* and TEX) demonstrated great intellectual as well as syntactical weight, and their influence on subsequent software cannot be overstated.

Between 1976 and 1980, Brian Reid developed the *Scribe* document preparation system (Reid, 1980b). This combined an author interface (editor) with a typesetter driver (compiler or processor) in what was still the traditional manner, but in the editor, it was *only* possible to enter and mark up text content, *not* to design the layout or modify it. Layout designs and changes had to be done separately, beforehand, by an expert, so that the author only had to write text and tag it with a semantic label; perhaps the first example of a **template** in structured text editing. These tags began with an 'at' sign and enclosed their arguments in square brackets; for example `@Heading[Table of Contents]`. The reason for binding the editor so closely to the document model and layout was to try and prevent authors implementing their own styles, and thereby breaking the publishing model of consistency (a habit we have already encountered in 'Interventional view of structure' in the list in section 1.1.3.1 on page 16).

In 1978, Knuth wrote TEX, which was 'only' a processor — no editor was included. The syntax used a backslash as the escape character and curly braces as delimiters for arguments. The semantics, however, were at the opposite end of the spectrum from *Scribe*, which imposed rules on what markup could be used. In TEX, authors could write their own markup (commands), and virtually any command could go anywhere; indeed any command, including the kernel of 300 or so 'primitives', could be entirely redefined. TEX also included a complete language for typesetting mathematics, substantially more compact and powerful than the facilities in *TROFF* or *SCRIPT*, and it implemented a model of typesetting which was æsthetically much closer to hand-set or hot-metal type than anything before. TEX proved immediately popular in disciplines with a strong mathematical content, but the complexity of the language and the lack of any semantic authorial overlay held back adoption outside those fields until the arrival of LATEX in 1986.[14]

[14]Apart from its typographic and markup features, the TEX processor was notable for being the first truly portable typesetting system. The original version was written in SAIL, but TEX82 was rewritten in WEB, a 'literate programming' language generating both Pascal source code *and* TEX

2.2.1.4 Wordprocessors

The advances made in macro representation and generalised markup initially failed to make the transition from the character-mode plaintext editor to the graphical user interface. GUI operating systems and applications were introduced on the Xerox *Alto* (1973) and *Star* (1981), and the Apollo *Display Manager* (1981). Related work led to the development of the platform-independent X Window System (1984, but based on an earlier system called W) and also to the Apple *Lisa* and *Macintosh* (1984); and eventually to Microsoft *Windows 3.0* (1990).[15]

Although the X Window System was common on Unix systems, Apple occupied only a niche market in colour and graphic interfaces in the 1980s and early 1990s. Commonplace office wordprocessing was mostly done with IBM-compatible PCs running Microsoft's MS-DOS applications on mainframe-style 80 character by 24 line screens. There was a huge range of wordprocessors available: at one end of the scale there were simple but effective shareware dot-line markup systems such as *PC-Write*, which could hide the markup and colour the displayed characters to indicate formatting (later versions used bold and italic screen fonts); at the other end there were more sophisticated corporate systems such as *WordPerfect*, which hid the markup in the body of the screen but provided a marching display at the bottom, where a telltale would indicate what markup was applied to the current cursor location.

Although graphical screens were widely available, they tended to be limited to non-text applications. Wordprocessing was still a character-cell console application, although it used menus and mice extensively. Most of the systems eventually developed a styling mechanism and the ability to operate macro-like facilities, but the macros were often simply bound to keystroke shortcuts, especially on the F1–F10 keys, which expanded to procedural markup, which got inserted in the document.

Text-handling applications (initially wordprocessors; later desktop publishing systems, below) enabled the spread of computer use into the hands of end users

---

documentation from a single master file. The availability of Pascal on a wide range of computers meant that TEX could run on even a modest office minicomputer. Although the WEB system was subsequently ported to many languages, the canonical version of TEX is now generated in the C language from Knuth's master source. C has a much wider penetration than Pascal, making TEX and LATEX uniformly available on almost every platform from smartphones to supercomputers.

[15] *Windows 1.0* (1985) and *Windows 2.0* were indeed GUIs but not widely adopted.

with no background in computing.[16] In the rush to market these products, features such as document structure, macro control, and generalised markup were not regarded as critical, perhaps because it is easier to sell something that looks pretty and appears to need only minimal training, than something that is more capable and durable but requires extensive learning. Product vendors, from individuals to large corporations, may also have felt that it was in their interests to cater for the end-user's lack of knowledge or experience. This goes a long way towards explaining the plethora of error dialogs, warnings, configuration menus, pop-up help systems, confirmation passwords, and floppy-disk interlocks which were developed, sometimes farcically inadequate, in an attempt to educate the user and explain the complexities of an interface which was originally designed to obviate the need for them (Cooper, 2004, Chapter 1). This despite the fact that the need for 'consistent and unobtrusive aids' (Relles & Price, 1981) had been known for a long time.[17]

Since the rise to predominance of Microsoft *Windows*, the wordprocessing market leader has been *Word*, although perhaps more on the back of bundling *Office* (wordprocessor, spreadsheet, and presentation software) with the operating system, rather than for any particular superiority in the product. While *WordPerfect* is still available, neither it nor *Word* has any structured-document capability in the general editing interface. In fact, both products come with a separate XML editor. *WordPerfect XML*[18] can read a DTD, and allows the user to create stylesheets to simulate WYSIWYG editing. The XML editor in *Word* reads W3C Schemas, not DTDs, and its interface is more cluttered and slightly less conventional than *WordPerfect XML*'s and the stylesheet interface more complex. Both provide most of the expected XML editing functions (see Table 3.10 on page 140 and Table 3.11 on page 150). Although *WordPerfect XML* has been reported in use in some XML-based research projects in the Humanities (see the archives of the `TEI-L` mailing list at `http://listserv.brown.edu`, *passim*), it has largely been superseded by other XML products.

This author is unaware of any significant use of *Word*'s XML editor for general XML editing, but there are numerous XML plugins for *Word*'s traditional and familiar

[16] The initial attraction of personal computers was to accountants, with the introduction of *VisiCalc* (1979), but text formatting in the manner of a wordprocessor was not initially a feature.

[17] It may be noted that the author's own institution still regularly receives requests for training courses in 'how to use a web browser' — not in the intricacies of configuration or customisation, but in how to click on blue, underlined links.

[18] Originally available as *WordPerfect SGML* from that era.

wordprocessor interface, enabled by the fact that *Word* now uses an XML-based file format. This is the complex and arcane OOXML, standardised (ISO/IEC 29500-1/4:2008, 2011) amid considerable public controversy and disquiet over the abuse of ISO procedures (Paul, 2008). As a result of a patent lawsuit by i4i (one of the plugin makers), the native XML editing component of *Word* was withdrawn, and is no longer available, although existing users can continue using it (Albanesius, 2010).

FLOSS wordprocessors became significant players with Sun Microsystems' previously proprietary *StarOffice* product being released as *OpenOffice* in 2000. This offered competition to *Word* and its related components in the office software market, but it too provides no structured editing facilities, despite being one of the few wordprocessors to use containment on list items in its markup. With Sun Microsystems now subsumed into Oracle Corporation, a 'fork' of *OpenOffice* called *Libre Office* has become more common, partly out of suspicion of Oracle's commercial intentions. As in the commercial marketplace, there are plenty of smaller FLOSS products, but equally with no structured editing. Despite using the XML-based ODF, also an international standard (ISO/IEC 26300:2006/Amd 1:2012, 2012), predating OOXML, neither *OpenOffice*, *Libre Office*, nor IBM's *Lotus Symphony Documents* (popular in offices using their *Lotus Notes* collaboration package), has a native XML editor.

An alternative approach for authors seeking structural guidance from software was the **outliner**, a once-popular tool which has largely fallen into disuse. An outliner used a simple hierarchical display, allowing the author to create the outline of a document in terms of chapters, sections, subsections, etc. It was possible to navigate around them, adding notes as ideas developed, and eventually to transfer the text to a normal wordprocessor. More complex outliners were plugins for a specific wordprocessor, with a more tightly integrated hierarchy and editing ability. Quint (1989, p. 41) cites a 1984 product, *ThinkTank*, as the first of its kind.[19]

[19] One seasoned author noted that writers accustomed to complex documents tended to think hierarchically in any case, and found that outliners did not provide the control they sought; those who did not have a hierarchical document model did not perceive the affordance of an outliner and sometimes used it as a flat text processor.

2.2.1.5 Desktop publishing

While desktop *typesetting* can be said to have arrived with TEX in 1978, its lack of ready-made page makeup abilities meant that it was restricted to the production of static page layouts suitable for books, journal articles, reports, and similar continuous-text document types. TEX was (and remains, even after its 1982 revision, and the later arrival of LATEX) unsuited to the newspaper, magazine, newsletter, advertising, and packaging layout models, where every unit of composition can have a different layout, and where text may be constrained to smaller unit areas placed on the page arbitrarily or according to a grid, or has to flow across larger unit boundaries, with a heavy emphasis on colour, graphics, and visual impact.[20]

Desktop Publishing (DTP) systems which could handle this material emerged in the mid-1980s, taking advantage of the increases in memory, disk space, and speed of PCs, and borrowing heavily from the GUI systems already in use. The absence of a standard GUI operating system on PCs meant that many DTP systems invented their own, which meant a substantial level of incompatibility in the way the interfaces worked compared with the unity achieved by the Apple Macintosh. As with wordprocessors, there were countless systems on the market, but the leaders for many years were Aldus Corporation's *PageMaker* (1985), long discontinued but still in use in some places; Ventura Software's *Publisher* (1986), still available from Corel Corporation; and Quark Inc's *QuarkXpress* (1987), still in widespread use.

While it was possible to use *Ventura Publisher* as an authoring system, it was designed more as a workflow component to allow authors to continue writing in their preferred wordprocessor, and to import the completed document for the final layout, and it is still widely marketed as such, with the ability to import XML documents. *PageMaker*, on the other hand, could certainly import text prepared in other systems, especially wordprocessors, but its editing interface was designed for authors and editors to write with, so it provided all the conventional functions found in wordprocessors, as well as its own *forte*, its page-layout and imposition features. *QuarkXpress* was an early market leader, targeted at the professional compositor, graphics specialist, and page-layout artist, with extensive (and highly accurate) facilities for colour separation and matching as well as page layout and typesetting. Extensibility was provided by 'Xtensions', plugins using an open API

[20] TEX is certainly technically capable of handling these models, but it would require excessive programming effort to cater for the wide variance and range of page layouts required.

which customers and users could develop and market; and the Tags file format, which enabled import from external sources. None of the products had any well-developed structural markup: the stylesheet facilities in all three products were aimed at making final visuals rather than preserving the structure of the document for reuse, although *PageMaker* was particularly strong in its support for long and complex documents, a field targeted by *Publisher*, but only matched by *QuarkXpress*.

PageMaker's [unrelated] successor is Adobe's *InDesign*, the current market leader, and the main competitor to *QuarkXpress*. *InDesign* has powerful XML import facilities, but is not in itself a structured document editor. Some of *QuarkXpress*'s extensions do now provide some structured editing capability, including XML import (see section 6.4.2 on page 365).

A different position is taken by Adobe's *FrameMaker* (1995, although earlier versions were available before Adobe took it over). This is an editor and formatter targeted specifically at the structured documentation market, with support for SGML and XML import (although with sometimes questionable export to the same formats). It also has an unstructured editing mode, which allows it to act as a conventional DTP system. The FLOSS product *Scribus* (2003) provides similar, but much less extensive, features to the main DTP systems but the file format is XML-based, although import and export does not have the same coverage as the commercial systems.

2.2.1.6 Structured editors

The markup concepts used in early formatters like *RUNOFF* became the underlying principles for subsequent generations of formatters and their markup syntaxes, including *Scribe* and TEX, and ultimately, if indirectly, SGML and XML (Goldfarb, 1996). They also to some extent influenced the development of LATEX, but although Lamport explicitly intended LATEX markup to represent the document's structure (Lamport, 1986), there was no intention for it to be independently parseable as with SGML: the only validation of a LATEX document is that it can be processed by TEX without error.

The file formats used for information interchange between editing systems and their formatters were of critical importance to the development of structured systems. Files created by early editors were mostly plaintext, as the objective was to be able to create program source code and data files which were openly

accessible. In wordprocessing and DTP systems, however, binary and proprietary formats became common, partly for speed and compactness on early hardware,[21] and partly from a fear common in business that competitors would steal a company's customers if the file format was easily accessible, because transferring the data would be too easy. It seems not to have occurred to these companies until very much later that *writing a better program* would be a far more secure defence against the poaching of customers than making the file format impenetrable.

A widespread market in file format converters therefore arose, using reverse-engineering to try to work out how a particular program stored its information, as well as published details on specific programs and formats (Born, 1995). The publication of *The GNU Manifesto* (Stallman, 1985) appears to have been the earliest ideological attempt to move away from file-format secrecy (Feller & Fitzgerald, 2001). While binary formats can be important for speed and compactness, this is less important with fast and capacious modern hardware and networking, and there is now a strong and compelling practical argument for the use of open, plaintext formats for master storage. Most editing systems can now use XML for some combination of import, export, storage, or transfer.

While the editing systems now used for structured documents derive many of their key concepts from early editors (mark, copy, paste, insert, etc), the dominant interface is now the composite WYSIWYG editor-formatter. In the sample of systems examined in section 3.2 on page 133, the majority are firmly based in this mode, although many provide a plaintext view as well. The significant exceptions are among the LaTeX editors, where only two systems had WYSIWYG interfaces (a third, *BaKoMaTeX*, was not included).

No attempt has been made to write the canon of XML and LaTeX editors: this would in any case be impracticable in a field still in development. The first SGML editors developed from implementations of GML such as that applied to later versions of *Scribe* as described in section 2.2.1.3 on page 77. Early commercial XML editors were derived from existing SGML products (for example, SoftQuad's *Author/Editor* and Arbortext's *Editor* [later *XMetaL* and *Adept*/*Epic*] respectively), or written from scratch by individuals or companies with a long history in the field (for example, STiLO's *Document Generator*). *Emacs* acquired an *sgml-mode* at

[21] There were many additional reasons for binary file formats, but one in particular concerned markup: *Word* and many other pre-XML file formats (`.doc`, for example) were based on **standoff markup** (**out-of-line markup**), where text was stored as uninterrupted strings, and the formatting stored elsewhere in the file, using character-location pointers to the positions in the strings of the characters where the formatting was to be applied.

an early stage, then *psgml-mode*, *xml-mode*, *xxml-mode*, and most recently *nxml-mode*. XML editors have been written to address many different markets using both conventional and novel approaches (Quint & Vatton, 2004), such as the editing of 'data' XML; XHTML for the Web; DITA, heavily used in industrial documentation and technical writing; incorporating an IDE for processing with XSLT; embedding XML in web pages; or simply editing general-purpose XML on the 'better mousetrap' basis.

LaTeX editors underwent a different process. Donald Knuth had released TeX informally as free software[22], with the caveat that only he could issue updates, and that only systems conforming to a strict set of tests could be called 'TeX'. Leslie Lamport followed this pattern with LaTeX (Lamport, 2000), later transferring control to the TeX Users Group. Early editors were either modal adaptations to existing editors (*Emacs*, *vi*, etc) or relatively simple menu-driven plaintext editors. Only a few commercial systems have appeared, but many of them are very active, such as *PC-TeX* and *Scientific Word*, and some are quite recent (*BaKoMaTeX*). They tend to ship complete systems with their own editors; other systems, such as the leading free Windows and Mac implementations, *MiKTeX* and *MacTeX*, ship with one of the many free LaTeX editors, and users can choose and change editor at any time.

The editors selected for analysis in this study are listed in Table 3.9 on page 136.

2.2.2 Usability and structured editing software

As text-editors in the early stages tended to be used for writing and maintaining programs and data, rather than structured documents, much of the initial work on editor usability was concerned with speed, user performance, and error rates, often using KLM techniques: Card (1978); Card, Moran, and Newell (1980); Bovair, Kieras, and Polson (1990); Toleman and Welsh (1994). However, a comparison by Roberts (1979) of four editors used for text-editing, while concentrating on speed and learnability, also included a short list of functions, but only for plaintext edits, not at the level of generic tags related to structured text. Only one of the original editors studied (*TECO*) is still available for modern systems. Nevertheless, Roberts' work led to a significant use of her methodology, and much re-testing was done with larger lists of editors, including *Emacs*, still in

[22]The more formal concepts of FLOSS did not become codified until after the establishment of the FSF in 1986.

heavy use today (Roberts & Moran, 1982). A later study replicated the experiment with another different set of editors (keeping *Emacs* for comparability) and identified some minor methodological flaws, but essentially validated the process (Borenstein, 1985).

As windowing environments started to become commonplace, O'Keeffe (1987) studied six editing tasks of varying complexity in both windowing and non-windowing systems. These were basically finding-and-typing tasks with some decisions needed on placement, and the study found that the benefits of windowing systems only became apparent in more complex tasks. While this study was only marginally related to document structure, it is significant that it was an *editorial* task that was selected with which to measure the speed and accuracy of working in radically different screen environments.

In an overview of document preparation systems undertaken in the late 1980s, Furuta (1992) separates syntax-directed *program* editors from *document* editors, formatting systems, and markup languages (my emphasis). The use of syntax-direction at the time was restricted to a few systems like *GRiF* and *Quill*, which are mentioned under integrated systems as 'WYSIWYG-like', as they functioned as both editor and formatter for structured documents. Significantly, Furuta does identify editorial control of document objects via the use of *styles* (his emphasis), relating the use of *Scribe*'s generic markup to the use of styling in early WYSIWYG wordprocessors:

> Styles are one means of combining flexibility similar to that of generic markup with a WYSIWYG form of document manipulation.
>
> (Furuta, 1992, p. 34)

The differences in usage of the phrase 'structured documents' to which we alluded in 'Literary view of structure' in the list in section 1.1.3.1 on page 16, and the distinction between 'data' and 'document' texts (see section 1.1.3.3 on page 22) mean that much work of apparent relevance in fact has application only to 'data' XML. There is a substantial body of work on grammar-based editors for structured *data* expressed in XML — for example Sifer, Peres, and Maarek (2002); Hu, Mu, and Takeichi (2004) — which contributes valuable insights into the grammar-based approach, but which can only be applied with difficulty to documents where mixed content is the norm.

A detailed analysis of what we would term the usability of the printed word is in

Southall (1984),[23] including the distinction between microstructure (flow) and macrostructure (hierarchy and pool, as in Abolhassani et al. (2003)). He identifies the author/designer conflict which we saw on page 17, and emphasises how important it is that

> [...] the hierarchical relationships between elements should be clearly expressed in terms of the graphic relationships of their headings; as should their relationships of containment and sequence. (Southall, 1984, p. 82)

(By 'elements' he means the structural elements of the document hierarchy.)

The technological constraints to which he refers (speed, resolution, and the restrictiveness of controlled document structures) are now largely solved. The two major problems he identifies (difficulty of use and 'miserable' graphic design) are — in our view — largely not. He attributes this to the mismatch between the visual, graphically-oriented approach of the classical designer and the algorithmic, symbolic, and abstract approach of the computer scientist (for which we can now read 'document engineer'). The pervasive graphical nature of modern interfaces may perhaps mean that today's graphic designer is only marginally better-equipped in terms of technological understanding than in the 1980s, having been shielded from the need to know how structured documents work. While graphic designers *can* now use [graphical] computer-based production systems, they have also perhaps become divorced from the underlying document structure.

The problems of authoring semantically-rich structured documents were considered by Quint and Vatton (2007) in describing a new methodology.[24] This relies on XHTML at the authoring end with the additional semantics carried in `span` elements mapping to *XTiger*, a semantic vocabulary in XML which the authors devised for the purpose. While this can achieve the goal of mapping authorial XHTML to a desired target XML schema, it does not address the usability of the editor interface used to create the document in the first place.

In a review of automatic document processing, Hurst, Li, and Marriott (2009) identify a 'shift of focus from micro-typographic concerns such as line-breaking to macro-typographic concerns such as page layout'. This may in part be due to the need for documents to be adaptable to more than one platform, but the authors

[23] Some details in the original paper were updated in a later *corrigendum* (Beeton, 1985).

[24] The application of post-authorial schemas to XHTML to achieve mapping to another format has been in use for some years, but the authors' formal description and the use of a custom dialect is a useful new approach.

also recognise the problem that dynamic document content (material retrieved in real time, as the document is requested) means that the final format of the document cannot be wholly predicted. Strict adherence to a formal structure lessens the effect of the problem, but factors such as float placement, table formatting, and the selection of grid layouts are still not fully resolved. On the matter of adaptation to smaller platforms, the authors identify techniques of scaling-down or specifically adapting certain page elements to improve usability, but give no further details.

A more practical approach is taken by Geers (2010), who highlights the changes in automated production which caused Southall's graphic designers so much difficulty. His experiments used the *Xopus* editor, which we include in tests (p 153), and tested a number of editing operations including our concept of drop-down semantic menus from font style buttons (in section 4.3.4 on page 253). His approach is a useful one, as it included testing a technique of intertextual semantics due to Marcoux and Rizkallah (2009), which displays semantic 'tags' alongside the formatted document element, which we have explicitly excluded in this research. He concludes that the hierarchical structure of the document is best 'not shown' (that is, not shown in markup, but only in formatting such as headings); and that an editor should be 'smart enough' to detect what the author is trying to accomplish. He also notes that 'a generic structured document editor is by definition less usable than one that is adapted to a certain schema' (Geers, 2010, p. 26) and that a schema for structured documents must not be 'too unconventional' if it is to be usable with a structured editor.

The growth in emphasis on embedding semantics into published text — the 'Semantic Web' (Berners-Lee, Hendler, & Lassila, 2001) — has created an advanced genre of editor targeted at the creation of online resources that can be searched meaningfully and harvested to populate ontological collections. In *RDFaCE*, for example, one view presents a standard WYSIWYG editor interface (based on the popular web-embeddable *TinyMCE* editor); a second view shows the embedded metadata, with edit panes for different forms of annotation; and a third view reveals the raw HTML, aimed at the document engineer or semantic domain expert (Khalili, Auer, & Hladky, 2012). While this approach allows great flexibility in adding annotations (its purpose), the use of ungoverned XHTML makes it applicable mostly for web pages rather than formally-structured text. It is clear, however, that usability is not always a major concern in this genre: in a later review of user interfaces for semantic authoring, the same authors criticise the lack of formal UI evaluation in the systems reviewed (Khalili & Auer, 2013b,

p. 14). Some very recent developments in editing with HTML5 and CSS3 are discussed in section 6.3 on page 341.

2.2.3 Design processes

At each phase in the evolution of structured editing systems, the first few editors are conventionally written by those working in the field. These are often proofs-of-concept, done for their own benefit, or for that of colleagues and other users, and typically (following the FLOSS model) distributed free of charge and free of restrictions, and the involvement of others is usually encouraged (see Feller and Fitzgerald (2001) and the brief descriptions in section 3.2.1 on page 134).

This is not simply altruism: developers of new concepts and new ways of doing things need functioning implementations to test and demonstrate, and the fastest way to do this is to make your own. In some cases the software may exist before the concept becomes concrete, in the form of an experiment or test, or as a personal project. Where companies are involved in the process (as for W3C development projects such as XML), the release of free software is often paralleled, or at least rapidly followed, by commercial implementations for early adopters within their customer base.

The design processes involved are therefore very disparate, and it is not possible to say at this distance in time, with any accuracy, precisely which philosophies were involved in early developments of editors. Some are known to have been written to implement a specific feature, such as real-time validation (SoftQuad's *Author/Editor*) or a real-time WYSIWYG view (Blue Sky's *Textures*). Others were developed out of existing non-SGML software for a specific customer base (Elsevier's *Pandora*), or had features added specifically to support them (SyncRO Soft added TEI support to *oXygen* for the academic market).

It is nevertheless clear that there were conscious efforts to support at least the developers' perceptions of the users' 'wants, needs, motivations, and contexts' (Cooper & Reimann, 2003). Despite the title of at least one paper, 'If SGML Is So Smart, How Come It Ain't Rich?' (Ensign, 1995), it is also evident that early SGML developers had a very good grasp of the business requirements of at least one market sector — after the US military made the use of SGML mandatory for vendor hardware documentation, editors were priced accordingly.

There is an interesting insight into the design decisions for *Scientific Word* and

Scientific Workplace in Mackichan (2007), particularly the decision to use XML as the internal storage format for these LaTeX editors. XML was chosen because it is 'easily converted to and from LaTeX' and because XML Binding Language (XBL) 'allows attaching behavior to (new) XML tags' (Mackichan, 2007, p. 336). While the second argument is undoubtedly true, many would perhaps query the assertion in the first that conversion *from* LaTeX is 'easy'.

However, given the context of this present research, we will be investigating why not all of the user requirements have yet been satisfied. The reasons behind this, in design terms, may lie less with the actual requirements of the users than with the perceptions of the market — that is, with the designers' informal models of who the users are. The use of personas in the design process has, in part at least, slowed the continual shifting of these informal models. With commercial software, from the vendor's point of view, the customer is typically the manager or procurement officer of the purchasing company. These customers may sign the invoice, but are unlikely to be the end users, and if they fail to buy what their end users need, then the vendor may never know (or even be interested in) the actual requirements. As we saw in section 2.1.2.1 on page 61, the rise of freely-downloadable software as well as Web-based direct sales of commercial software, particularly the 'try first, buy later' mechanism which is now commonplace, is radically changing this perception (Nielsen, 2000).

2.3 Structured documents

In summary so far, there are six factors which need to contribute to the accessibility of the document structure in an editor:

1. design methodology (UCD);
2. affordances;
3. intuitiveness/obviousness;
4. design processes;
5. document type design;
6. typographical formatting.

There are techniques such as Smart Insertion (SI), **ghosting** (Figure 4.9 on page 248), and Target Markup Adoption (TMA) which are implementation-specific, and may span two or more of the factors above.

In a report on a panel session[25] at the 2007 XML conference, Fahlgren (2007) reports a question answered by Robbert Broersma of *Xopus*:

> 'How can we make XML editor UIs less confusing (b, i, u buttons) for people who don't know XML? Many authors find the *behavior* of XML tools broken.'
>
> That is *exactly* what *Xopus* does; it allows you to edit any XML format as intuitively as editing in *Word*, or the like. Just specify what elements should behave as paragraphs, emphasis, sections, lists, tables, etcetera; and all the shortcut keys and toolbar buttons you know from *Word* come to life in your browser.

'Just specify' is perhaps a little oversimplification, but in principle, *Xopus* does provide a semantic mapping within the UI between the familiar interface components of a wordprocessor and the specific element types of an XML vocabulary.

An additional comment from one of the panel (Mark Jacobson) claimed there were 'two non-technical challenges for XML editors: many people simply want to use *Word* and have no interest in adding structure.' This is undoubtedly correct, whether those users' reasoning is correct or not; but we are concerned here expressly with those who *do* have an interest in adding structure. It is arguably

[25] Mark Jacobson, Charlton Barreto, Jeff Deskins, and Laurens van den Oever: *Where are XML authoring tools today, where are they going, and what do we want? What do authors, editors, and copy-editors actually need to do their work?*

the task of the excellent designer to 'purify the dialect of the tribe' (Eliot, 1942, II) by providing designs with foolproof but rich affordances.

More recently, there has been a growing view expressed that the historical path of managed editing for structured documents is overkill for most authors.

> I want structured text, but authors don't want to write structured text (and why should they?). Every single structured authoring tool I have seen gets in the way of the author. (N. Gibson, 2013)

Production teams now routinely convert styled *Word* documents to a formal (XML) structure. This method may well be applicable to an even larger class of documents than at present, but this view overlooks the advantages of 'early binding' (getting the material into a structurally-reusable format as early in the production process as possible).

As mentioned in section 1.1.2 on page 8, the application of formal methods to the analysis of literary structure in computerised text was first treated during the 1980s, although previous work on editing hyperlinked text can be traced back to the File Retrieval and Editing System (FRESS) and the Hypertext Editing System (HES) (DeRose, 1997, p. 149), and earlier work inspired by Ted Nelson's Project Xanadu. The use of computer techniques in the Humanities, where there are major contributions to the development of textual theory, dates back to the work of Busa (1980).

The important developments in the technical field in the 1960s and 1970s with text processors for handling formal document structure (described earlier in this chapter) led to the widespread availability of document production systems of differing types, power, and complexity. The prevalent systems for handling formal structured documents (SGML, LaTeX, and XML) have largely followed a divergent development path from the wordprocessor and DTP systems described in section 2.2.1.4 on page 79 and section 2.2.1.5 on page 82, although there is an increasing awareness of the importance of structure in those systems for handling unstructured and semi-structured documents.

2.3.1 Technical writing

Technical writing was one of the earliest applications of computer editing and formatting (see section 3.2.1 on page 134) so it was natural that advances in the

use of computers in writing should have been most closely documented in that field first. It is also often the case that the technical capabilities of the software are explained and discussed using the very software that is under discussion. This can be very useful to the authors of the document and the software, as it means that the document itself is exercising the software it describes, by way of proof or validation; but it is also useful to us, as it provides evidence of the conscious choice of certain ways of working as well as their effectiveness.

However, while it is clearly possible to use XML or LaTeX for technical writing, it is uncommon outside documentation on XML or LaTeX themselves, and a few closely-related areas such as mathematics, engineering, and science. Informal enquiry among members of the Society for Technical Communication (STC) indicated that *FrameMaker* and *Word* appear to be preferred for actual writing, with *InDesign* and *QuarkXpress* used for page makeup and layout. The reasons claimed included familiarity of the interface, easy learning curve, low setup cost, and flexibility in handling multiple document types. The proportion of technical documents being authored in XML editors such as *XMetaL*, however, is perceived as being on the rise,[26] as is the technique of authoring in one system (for ease of use) and then converting to the target format with a processing tool.

This is a well-used technique elsewhere: with structured document systems it is commonplace to author or edit in an XML editor, and convert using XSL to Formatting Objects (FO) to PDF, or with XSLT to LaTeX to PDF. By doing so, you keep the content in a structured system as long as possible, so that it remains under programmatic control for the purposes of document management. This has its counterpart at the start of the production process too: what Strange (2003, p. 157) refers to as the **XML-in** route, whereby content is encoded in XML as early as possible in the process for the purposes of control, even if the markup format is not the one used for final production. However, strict control must be observed to avoid 'tweaks' to the appearance being inserted later in the process without being inserted in the master document also.

2.3.2 Document type design

We explained in section 1.1.4.1 on page 27 that a DTD or schema can be used to describe the existence of the named components of a document and the

[26] These editors were referred to as 'next-generation', although they have been around for over 20 years.

relationships between them. This has two purposes:

- it guides the formation of an XML document, helping the author adhere to the structure and avoid errors in markup syntax, and enforcing the rules that certain elements can appear only in certain contexts;
- it may also be used in processing a document, to check the structure against the rules (**validation**) before going any further, and to acquire any default values for attributes that may not have been specified explicitly.

The term **syntax-directed** is often used to describe editors using such a predefined structure, but adherence to the markup syntax (pointy brackets and ampersands, or backslashes and curly braces, etc) is only part of the function. The enforcement of the structural rules means following a pattern: for example, a `<chapter>` element type is usually inadmissible (indeed, non-existent) in a journal article because articles do not have chapters: their primary structural division is the section.

The naming of the element types in a DTD or schema is critically important because at most points there is a choice: each time an author finishes (let us say) a paragraph, the next item may or may not be another paragraph — it may need to be a list, or a quotation, or a figure, or a new section. Conventionally, structured editing software has required authors to know the names of the element types or commands because they have to select them from an Insert menu. The names therefore have to be recognisable for what they are: that is, they must afford their implied utility. If an element type called `para` was in fact a list, the result would be endless confusion.

This has been a point of debate for many years among markup experts. There is a widespread view, elegantly summarised recently by Holman (2013), that naming markup after its intended use is wrong, and that it is an error to infer meaning from markup, no matter what was originally implied by the designer. This idealism is unfortunately misplaced: to take it to its logical conclusion, markup names would be like `A001`, `B001`, etc, much like early variable names in computer languages, and offer no clue as to their meaning or usage. A later comment in a private XML mailing list offered a less absolutist approach:

> So I think the answer is that `<para>` has no semantics that you can reliably infer *solely* from the choice of name, but at the same time the name was deliberately chosen so that your guess at the semantics has a reasonable chance of being right. (Kay, 2013, my emphasis)

We would go further and argue that the idealism of avoiding semantics is inappropriate in the context of usability, and that in addition to syntax direction, DTDs and schemas must provide semantic direction. This is a concept very close to that of Cognitive Direction ('Erkenntnissteuerung', see Enenkel and Neuber (2004)), which describes how the design of a book can govern the way in which it is used by the reader (Delsaerdt, 2011, with reference to early dictionaries). In designing a DTD, the document type designer must be aware of how the users will react to the choice of names, so there is a strong argument that the usability of editing software is to some extent affected by the usability of the DTD, where this is exposed to the user.

Some early DTDs deliberately used heavily-abbreviated names (`TI` for title, for example), partly from convention, but mostly to minimise typing in the days before macros and menus. This is no longer necessary, and there has been a tendency in the *data* field of XML usage to allow the concatenation of database, table, and field names as element type names, on the grounds that no human will ever see them. While this is largely true for machine-to-machine processing, it does reduce the usability of the schema for the document type designer. It has sometimes been asserted that the XML Specification's goal 'terseness is of minimal importance' (Bray et al., 2000, Goal 10) impinges on name length, but Jelliffe (2008) makes it clear that this was a goal for the designers of XML itself, not for document type designers.

We aim to test whether or not an interface can achieve the recording of markup without the need for the author to learn the names — whatever they may be[27] — by exposing their function in the interface instead.

2.3.3 Typographic design

In any synchronous typographical (WYSIWYG) editor, the only evidence of markup is usually the appearance of the content. By definition, any pointy brackets, ampersands, backslashes, curly braces, or other markup syntax items are not displayed, although some modes permit a telltale such as a marching display at the bottom or a style margin at the side. Following directly from Furuta's comment about styles (section 2.2.2 on page 86), it is clear that accurate styling

[27] It is noteworthy that the TEI is the only DTD this author is aware of that has explicitly enabled element types to be renamed (into another language or synonyms) while retaining the ability of programs to process the documents.

of each element type in its context is important for the usability of such an interface, as it provides the principal means of conveying to the user what type of element each one is. While ambiguity may be acceptable in an informal document, it is misleading when the reader has only the formatting to rely on as a guide to meaning, importance, relevance, or relationship to other parts of the surrounding text.

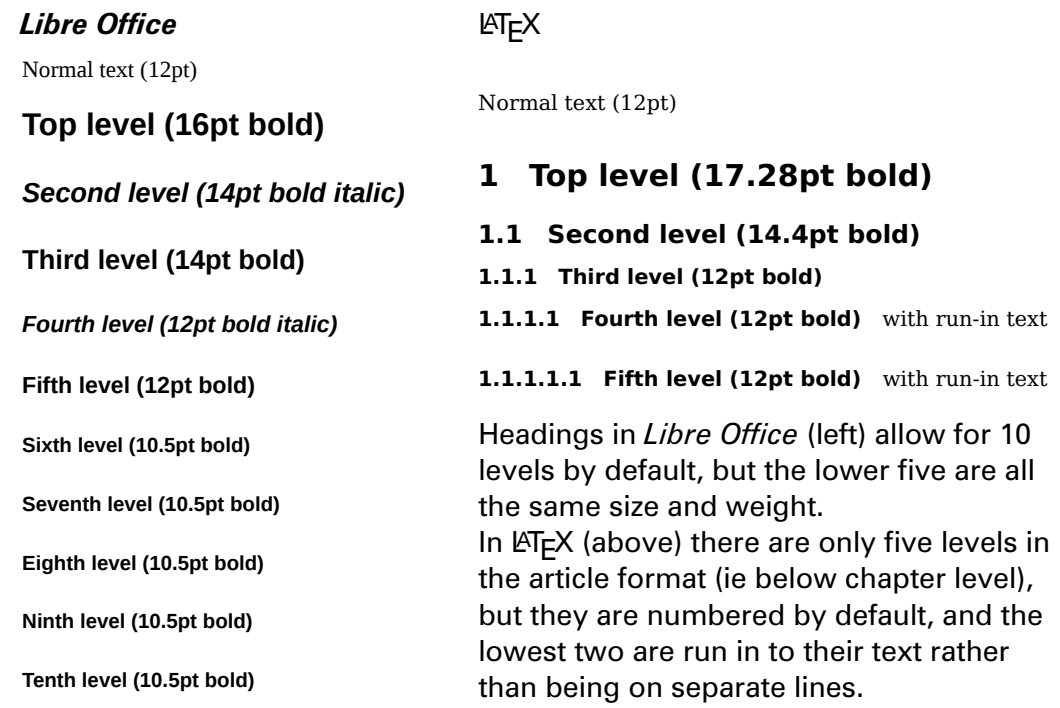


Figure 2.3: Implicit markup by type size and weight

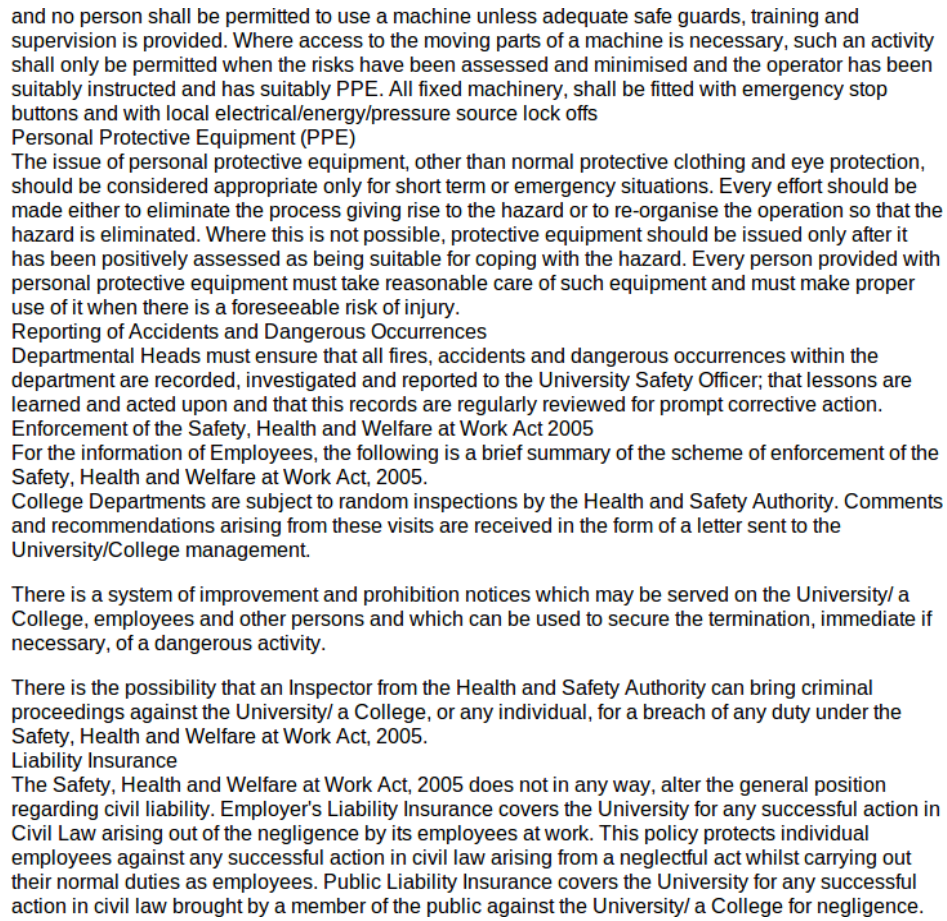
Figure 2.3 shows the typographic representations of the default section heading styles in *Libre Office* and LaTeX. While both of them can easily be restyled, the defaults illustrate that the designer only felt it useful to provide distinct styles down to the fifth level — understandably, as very few types of document would ever need to go to a finer level of granularity than this.

A similar effect can be seen in the way in which list items are represented: bullets of varying sizes and shapes, and numbering systems using digits, letters, and roman numerals, according to the depth of the list. Outside specialist areas like legislative documentation, it would be rare to use more than five levels of listing, and this is typically the limit to which default styling is available.

Given the importance of the typographic representation in conveying the significance of textual and graphical components, the editorial style designer may sometimes be in conflict with the typographic designer of the finished document.

This occurs especially when the author or editor needs to provide complex metadata or formal semantic markup, and can sometimes be resolved by multiple modes, panes, views, callouts, and other visual means (Khalili & Auer, 2013a, Figure 4).

It is unfortunate that some of the conventions of modern typographic design tend to emphasise indistinguishability as a virtue, as this makes it harder to implement style-driven editing. An extreme case is shown in Figure 2.4, where the formatting for the subheadings is identical to that of the normal paragraphs, and the hypertext links are not highlighted in any way (they are only visible when the mouse is passed over them). The only evidence of markup is the separated paragraph — perhaps suitably, as it concerns being fired for dangerous activities!



and no person shall be permitted to use a machine unless adequate safe guards, training and supervision is provided. Where access to the moving parts of a machine is necessary, such an activity shall only be permitted when the risks have been assessed and minimised and the operator has been suitably instructed and has suitably PPE. All fixed machinery, shall be fitted with emergency stop buttons and with local electrical/energy/pressure source lock offs

Personal Protective Equipment (PPE)

The issue of personal protective equipment, other than normal protective clothing and eye protection, should be considered appropriate only for short term or emergency situations. Every effort should be made either to eliminate the process giving rise to the hazard or to re-organise the operation so that the hazard is eliminated. Where this is not possible, protective equipment should be issued only after it has been positively assessed as being suitable for coping with the hazard. Every person provided with personal protective equipment must take reasonable care of such equipment and must make proper use of it when there is a foreseeable risk of injury.

Reporting of Accidents and Dangerous Occurrences

Departmental Heads must ensure that all fires, accidents and dangerous occurrences within the department are recorded, investigated and reported to the University Safety Officer; that lessons are learned and acted upon and that this records are regularly reviewed for prompt corrective action.

Enforcement of the Safety, Health and Welfare at Work Act 2005

For the information of Employees, the following is a brief summary of the scheme of enforcement of the Safety, Health and Welfare at Work Act, 2005.

College Departments are subject to random inspections by the Health and Safety Authority. Comments and recommendations arising from these visits are received in the form of a letter sent to the University/College management.

There is a system of improvement and prohibition notices which may be served on the University/ a College, employees and other persons and which can be used to secure the termination, immediate if necessary, of a dangerous activity.

There is the possibility that an Inspector from the Health and Safety Authority can bring criminal proceedings against the University/ a College, or any individual, for a breach of any duty under the Safety, Health and Welfare at Work Act, 2005.

Liability Insurance

The Safety, Health and Welfare at Work Act, 2005 does not in any way, alter the general position regarding civil liability. Employer's Liability Insurance covers the University for any successful action in Civil Law arising out of the negligence by its employees at work. This policy protects individual employees against any successful action in civil law arising from a neglectful act whilst carrying out their normal duties as employees. Public Liability Insurance covers the University for any successful action in civil law brought by a member of the public against the University/ a College for negligence.

Web page, author's collection (since updated)

Figure 2.4: Typographic design using indistinguishable styles

In conventional editing (ie *not* the specialist semantic editing described above), the typographic rendering which can do duty for markup is composed of:

- Typeface (Times, Garamond, Helvetica, Univers, etc);

- Font weight and shape (bold, italic, and other variants);
- Size (including linespacing and character-spacing);
- Colour (foreground and background);
- Vertical spacing (above and below);
- Horizontal alignment (margins, indentation);
- Positioning (subscript, superscript);
- Location (left, right, centre, margin, header, footer);
- Generated text (bullets, numbering, prefixes, suffixes).

While these would conventionally all be specified in the stylesheet or template, authors appear convinced that they should have the arbitrary power to change the publisher's formatting by themselves, as mentioned on page 17. This is usually overcome by allowing the author to modify the appearance in their *local* stylesheet while insisting on retaining the names of the styles or elements, so that subsequent processing can use the authoritative or 'house' stylesheet. This approach also has the advantage that authors needing larger type or other features to aid visibility while writing or editing can make such changes without affecting the final result.

2.3.4 Interface guidelines

There is an extensive literature on interfaces at the level of detail of the individual widget (a graphical object with a function, either display or interaction). All three major operating systems maintain their own interface guideline recommendations or requirements, which — amidst the wealth of advice and requirement for programmers and interaction designers — have detailed explanations of when, where, and why an interface designer should, may, or must use a particular widget (Benson, Elman, Nickell, & Robertson, 2012; Apple Corporation, 2012; Microsoft Corporation, 2010).

The guidelines do not go down to the level of detail required for structured editing. In particular, because they deal only what is visible on-screen, applications which cannot distinguish markup from content will be unable to show the user that distinction, regardless of what the guidelines say.

2.3.5 Synchronous typographic editing

The kernel of this inquiry is the business of editing a formally-structured document, using direct intervention in a typographically synchronous display, with no visible markup. As described in the software analysis in section 3.2 on page 133, there are many programs which approach this ideal, but the demands of users in the analysis of requests (section 3.3 on page 161) and the survey of users (section 3.4 on page 176) make it clear that the programs do not completely satisfy these requirements.

In response to discussions concerning the direction this research was taking, an impromptu and informal but rigorous inquiry among delegates to the XML 2007 conference in Boston, which elicited the following responses to the list of requirements, which was then only in draft form (here in order of preference for implementation):

1. dropdown menus on B/I/U buttons;
2. no markup visible or referenced anywhere;
3. use of a framed (boxed) model for element mixed content;
4. standardisation of what the Enter key should do;
5. avoidance of the TAB key;
6. no font dropdowns (name, size, etc);
7. stylesheet generation by schema evaluation (*à la Epic*);
8. a 'new document' dialog;
9. prewritten generic styles for the popular DTDs;
10. mathematics-to-TEX and *vice versa* (see footnote 5 on p. 360);
11. available in-browser and stand-alone;
12. able to derive skeleton structure from imported well-formed XML (like *Fred*, see section 6.3.9 on page 354);
13. interface 'must look like *Word*';
14. must allow authorial deviation from style and structure, albeit with reluctance, and with suitable warnings.

There are several layers to the representation of formatted text:

Stream: At its simplest, an undistinguished stream of characters, typed by the author, can relatively simply be represented on the screen in the typeface, size, and style specified by the stylesheet. This would be the case for normal continuous paragraphic text, and is implemented in all 'WYSIWYG' editors. An even simpler example is the text entered in a plaintext editor such as

Notepad, *vi*, etc.

Embedded stream: A marginally more complex requirement is the representation of simple inline markup, which is just a single transformation of the content from its surrounding style to another, usually by adducing italics, bold, or some other typographic distinction.

Structural break: A third level involves the separation of the text from what went before and what follows by the introduction of a line-break, vertical white-space, and possibly some horizontal adjustment (indentation, centering, left- or right-alignment). Within this, however, the text itself is still treated as a stream (above). A structural break occurs between paragraphs, list items, quotations, and other block elements, as well as between hierarchical divisions.

Alignment: Tables, as Knuth notes, cost extra (Knuth, 1986, p. 231). The complexity of lining up rows and columns requires simultaneous horizontal and vertical positioning, as well as the stream behaviour once the content is handled.

Generation: The most complex requirement by a long way is the creation of additional text according to rules (such as those in LaTeX macros or in XML/XSLT transformations). This can include bulleting or numbering systems, cross-references, citations, part-names from technical databases, the expansion of abbreviations, and a host of other material (we also discuss this in section 4.3.6.2 on page 273). Without a rigorous link between what is typed and what actually appears, a fully-editable display is virtually impossible, because some of the displayed text is never typed, but created by the computer from additional information stored elsewhere. This is succinctly explained by Laurens (2007) with reference to remanence in the editing of LaTeX text destined for PDF output.

Layout: The finished or part-finished document is assembled from its component parts, as typed, into pages, and any floating elements (tables, figures, sidebars, etc) are positioned. While this also requires each element to remain attached to its input, in order to remain editable, it is one area where perhaps more intervention is required *after* typing the text.

In a synchronous typographical editor, all or most of the above are going on all the time: every keystroke typed causes the context to be reassessed, and rules applied to determine if the displayed document needs to be changed. Even in

normal typing of continuous text in a simple wordprocessor, there is the constant handling of line-breaks as the text reaches the right-hand margin, perhaps hyphenation and justification decisions, and then the repositioning of the cursor to the left-hand margin for the next character. In a more sophisticated editor, adding text in one place may cause text already entered elsewhere to be repositioned, for example to avoid a heading being left isolated at the bottom of a page: it has to be moved to the next page along with its first few lines of text.

Mathematics, music, and other symbological notations, while they require extra fonts and positioning, may be regarded as special cases of embedded streams or structural breaks, depending on their nature.

While some of the more powerful editing systems can be specially programmed to do a large amount of the above, it is not the default position: there is no system which does it all right out of the box (Anghelache, 2004).

2.4 Summary

We have traced the development of usability as a discipline from its parentage in postwar ergonomics, through several significant shifts in approach, to the current model of UCD and its effect on 'Web 2.0' interfaces. In particular, we examined the relationships between intuitiveness (or intuitability) and cognition, and between affordances and obviousness.

Before discussing affordances, we introduced the concept of users' experiences, which we termed UUXX, to explain the effect of accumulated knowledge from past exposure to IT systems. In examining the terms under which affordances have been treated, we narrowed the definition for this research in order to take into account both the efforts of the programmer/designer and the variability of users' past experiences. This allowed us to consider the methodology of usability testing and measuring in the light of UUXX.

In looking at the development and critique of editing software, we noted a wide variety of approaches leading to the separation of editors for programming from editors for writing documents, and subsequently to the evolution of structured editors separately from unstructured ones. These developments paralleled the technological changes in computing, but have left the separation between graphical design and systems design largely untouched.

Only very recently has the concept that the structured editor must show the structure explicitly been seriously challenged. The demands of technical authoring, document type design, and typographic design have all contributed to the processes involved in making a structured editor, but the emphasis has been on representation and customisation rather than the effect of the interface on the user.

CHAPTER 3

Data collection and analysis

1. EXPERT SURVEY — Objectives and design — Sample selection and administration — Analysis — Conclusions. 2. SOFTWARE ANALYSIS — Background — Software selection — Objectives — Methodology — 2005 results — 2009 results. 3. REQUESTS ANALYSIS — Objective and methodology — Procedure — Refining the sample — Categorisation — Analysis. 4. USER SURVEY — Questionnaire construction — Survey administration and processing — Analysis and results. 5. CONCLUSIONS.

It is a capital mistake to theorize before one has data. (Doyle, 1892, p. 163)

In the preceding chapters we have discussed some of the conceptual and technological constraints affecting the adoption of structured document methods. Anecdotal evidence suggests that authorial resistance to the perceived complexities of XML, LaTeX, and similar systems is visible in the habit of continuing to use unstructured systems even while acknowledging the proclaimed benefits of structured systems.

Empirical evidence is hard to find, perhaps partly because in organisations where a structured document system is being introduced, author rejection or hostility would be seen as signs of managerial failure, and thus not publicly admitted (this is certainly the case in more than one institution with which this author has had professional dealings).

The technological constraints of any change can be very significant at the organisational level, affecting the way business is done and requiring re-training

and re-tooling. The conceptual constraints are more subtle, requiring people to change work-practices and even opinions.

In order to evaluate these constraints we needed to gather sufficient data to provide an empirical basis for the investigation. Constraints at the organisational level (for example, 'doing business') were outside the scope of the study, but constraints on the way users use their software were seen as key to the task. In all, we considered that we needed to identify at least 14 factors which appeared to affect the way in which editing software is used:

1. the criteria for selecting a structured document system;
2. how such systems are perceived by the selectors;
3. what specific features are present or absent (required or not required);
4. the views of designers and implementers;
5. degree of choice (extent of available software);
6. scope of the functions offered by the software;
7. ease of use of those functions;
8. level of user demand for the software;
9. features specifically requested in that demand;
10. types of editor actually in use;
11. degree and type of use of the editing functions provided;
12. demand for functions to be changed;
13. demand for additional functions;
14. views of users on software recommendation.

In practice, some of these were effectively unobtainable (for example, the canon of all available software), and in some cases, the data were not available due to business confidentiality (specific features required by a company or consultant).

The factors appeared to fall into three main categories: *a*) business process and pre-purchase factors; *b*) availability of software and functions; and *c*) levels and methods of usage. As software purchase or acquisition is carried out both at a personal and corporate level, it was decided to split the first category into user

queries and professional advisory branches. Four investigations were therefore planned to measure these factors:

Expert survey (factors 1–4): Design and implementation of systems for businesses is typically done by experts, either in-house, or from the vendor, or by using external consultants. Even personal adoption may often follow from professional recommendation, as well as from exposure to advertising (a form of professional recommendation, no matter how partisan). A survey of expert practitioners at this level was conducted in order to find out what problems they encountered when specifying (or recommending), selecting, and implementing editing software for their clients (section 3.1 on page 107).

Software analysis (factors 5–7): The technological constraint factors were investigated by an analysis of the functions of the available software (section 3.2 on page 133). This was designed to measure two things:

1. the ability or otherwise of the software to perform the tasks necessary to the creation and editing of structured documents in XML and LaTeX. This represents the 'Functionality' aspect of the tripartite model proposed in McNamara and Kirakowski (2006).

2. the effort required by the user to identify and use those functions (the 'Usability' aspect)

These measurements were performed without reference to any set of requirements other than those mandated by the two file formats, and those known to exist from empirical sources already discussed in section 1.3.2 on page 45 (Plotnik, 1984; Bly & Blake, 1982; Grafton, 1998; Knuth, Larrabee, & Roberts, 1989; Legat, 1986; Neumann, 2001; Tannahill, 2008; Turabian, 1973; Walsh & Muellner, 1999; Williams, 1992, 1990; Baskette, Sissors, & Brooks, 1986).

Requests analysis (factors 8 and 9): The problems of asking users for their views have been discussed in section 1.3.2 on page 45. Fortunately, there was a readily-available corpus of voluntary requests and comments in both major fields (XML and LaTeX) in the archives of online forums. While this is necessarily a self-selected sample of the user (or intending user) population, it is by its voluntary nature substantially free from the difficulties of contamination by suggestion. The conduct and analysis of this investigation is in section 3.3 on page 161.

In order to perform a robust search of such a corpus, a framework had to be constructed and validated from the data gathered in the Expert Survey.

User survey (factors 10–14): The selection and use of software by a user is subject to a number of constraints, including personal preference, recommendation by a colleague or expert, institutional requirements, and formal methods. Formal qualitative and quantitative methods abound: every large company, institution, professional body, government department, and standards authority has them in one form or another, and there are many commercial offerings (Bandor, 2006; Goldfarb, Pepper, & Ensign, 1998; Maler & el Andaloussi, 1999). However, once software is selected and put into use, any difficulties encountered require the attention or advice of the vendor, software author, or an in-house or external expert. Formal selection methodologies are intended to help minimise difficulties after implementation, but the formality of the process may also inhibit participation.

Vendors and software authors (except perhaps in the case of FLOSS) are notoriously unlikely publicly to share details of the shortcomings of their products, and may be legally constrained from doing so. In order to obtain an overview of difficulties experienced, a User Survey was undertaken (section 3.4 on page 176), concentrating on the functions of the software actually used in practice.

[A preliminary report on the findings of the first three investigations appeared in a research note presented to the 2006 Extreme Markup Conference (Flynn, 2006). Some further work, and the preliminary findings of the User Survey, was presented in an additional note to the 2009 Balisage Conference (Flynn, 2009). A final note on findings and work remaining was presented to the 2013 Balisage Conference (Flynn, 2013).]

3.1 Expert survey: the starting position — what the informed experts say

This survey was used to gather baseline data on the professional recommendation and implementation of editing systems. It was administered to a group of XML and LaTeX experts who had extensive experience of editing software, both as software users and as systems designers and consultants. A pilot survey was conducted in July 2003–4 and the full survey in August 2004–5. Anonymity was guaranteed, so no personal details were recorded, in order to encourage explicit responses.

3.1.1 Objectives and design

The objective was to determine the views of expert practitioners on editing software they recommended, the adequacy or otherwise of the software they had encountered (with specific reference to deficiencies they had noted), special features they found important, and their expectations of the software. A secondary objective was to refine the questions for use in the User Survey (section 3.4.1 on page 185).

The general purpose of this research was described to the participants by way of introduction, but the specific details were not discussed beforehand.

The goals were to identify *a*) what methods expert consultants used for software selection; *b*) what they actually recommended; and *c*) what features or facilities persuaded them for or against any particular solution.

The survey was constructed to gather the following items of information:

1. Background variables were recorded to allow analysis of whether they could be associated with the respondent's approach to software recommendation.
   (a) Occupation, the respondent's job title or function;
   (b) Affiliation, the type of organisation[s] with which the respondent is associated (by employment or membership);
   (c) Operating system experience, representing the major platforms on which structured document systems have been available. An attempt was made to weight these, but this was abandoned in view of the predominance of the three major desktop systems and the time elapsed

since Microsoft's Disk Operating System (MS-DOS) and the others ceased to be viable platforms;

(d) Structured document markup experience:

i. Types of markup system used, including the major progenitors of current systems;

ii. Years of experience, banded with a breakpoint at the 10–year mark (relative to the date of the survey) when XML first appeared. Respondents with fewer years of experience are unlikely to have had any substantial involvement with earlier systems (notably SGML);

iii. Types of application used, covering the four broadest application groups, including two which are explicitly *not* the subject of this research ('data' and e-commerce applications, used to filter less relevant experience).

As this was an administered survey, the analysis questions were open-ended (with one exception at the second item in Q.2 below) in order to allow the participants latitude in expressing their responses:

2. Selection criteria;
   (a) What are the criteria you use to decide which editor is best suited to a given project?
   (b) What would be your *personal* choice of product for the following tasks for *a*) writing (authoring); and *b*) editing (someone else's work) (assuming XML, but otherwise you have a completely free hand to choose, ie unconstrained by cost, platform, or anything else)? [list shown to subject].

A newspaper or magazine article on the popularity of the Personal Digital Assistant (PDA)
The Owner's Manual for a washing machine
A romantic novel (for example, Mills & Boon)
Documentation of your own code
Your memoirs
Maintenance documentation for a ski-lift
An article on Perl for a conference or journal
A picture-and-story book for ages 4–8
A book on the architecture of Oxford

The leaflets included in consumer electronics purchases

3. Are there products you would prefer to use but cannot (for reasons of price, licensing, availability, platforms, etc)?
4. Can you identify up to three things your preferred software does which you consider marks it out as specially useful?[Respondents were asked to indicate if these were *Features* (defined as something the vendor believes is special), *Functions* (defined as facilities required in order to operate), or *Behaviours* (defined as ways of doing things).]
5. Can you identify up to three things about any of the structured-document software that you have tested or used which you consider are particularly poor?[Respondents were asked to indicate if they believed these to be Bugs, Deliberate but flawed design choices, or Carelessness.]
6. Are there any other facilities in which structured-document software you have experience of is specially lacking?
7. While using a structured-document software product, have you ever failed to identify how to do something the product was actually capable of?

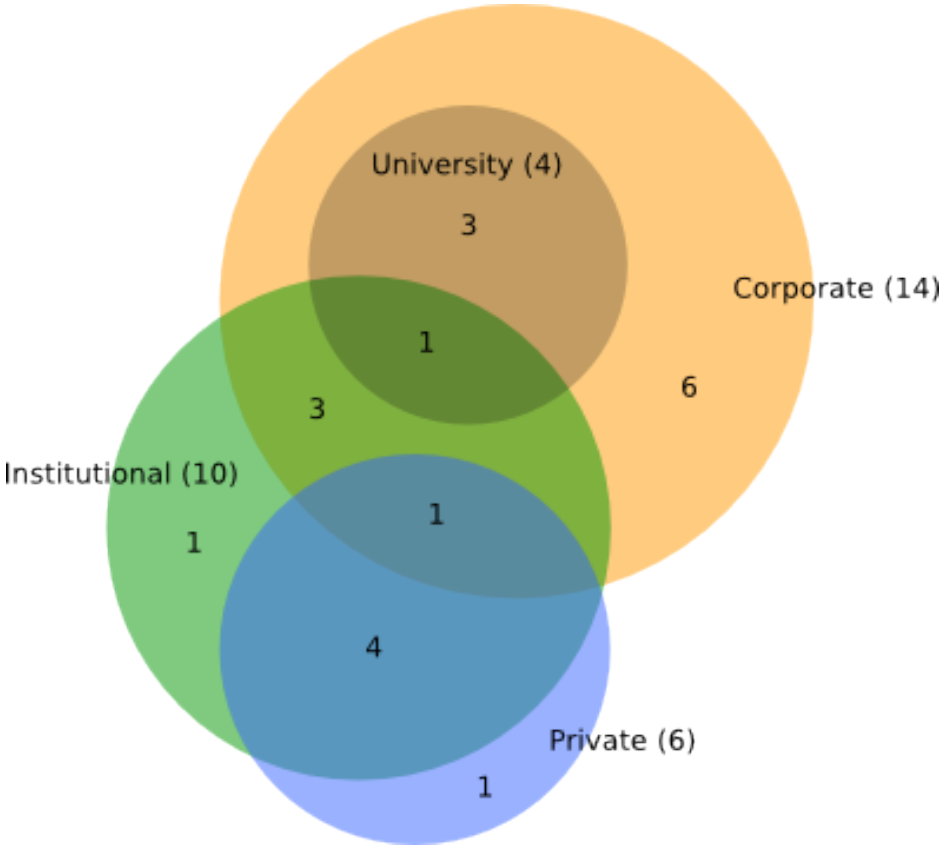
The tasks in Q.2b were chosen to represent common document publishing (authorial and editorial) tasks, but also to elicit any contrast between the expert's presumed knowledge of *a*) which document types probably *ought* to be handled using a structured editor, but which commonly are *not*; *b*) those for which it might arguably be overkill; and *c*) those for which it is probably essential.

The full questionnaire sheet is reproduced in appendix A on p. 371.

3.1.2 Sample selection and administration

The participants for the survey were invited from among the speakers and chairs at selected events in the structured document practitioner field in 2003–5: the XML Conferences in Washington, DC; the Extreme Markup Conferences [now Balisage] in Montréal; the TEX Users Group Conferences in San Francisco, CA, and Chapel Hill, NC; and the XML Summerschool in Oxford. This selection was felt to represent the user-orientation we needed for a user-oriented study: there are many other technical and academic events such as the Association for Computing Machinery (ACM) symposia on document engineering, and the TEI and digital humanities conferences, where structured document editing is an essential component but only a part of their scope, and approached from a different viewpoint.

The participants were chosen on the basis of their experience in advising users and organisations on the selection and use of software for structured documents, either as independent consultants or as staff members of organisations acting as consultants. Participants were guaranteed anonymity, as they were encouraged to speak frankly about the users and software they work with. The types of organisation represented by the participants are shown in Figure 3.1.



Calculations: Chris Seidel's Venn Diagram Generator; Graphics: Author

Figure 3.1: Expert survey: Q.1 Participant affiliations

The questions were piloted with six participants, on the basis that they would agree to a second session with a revised set of questions. All agreed, but it proved unnecessary to hold a second round. Two participants declined to participate (being restricted from doing so by their employers) and were replaced. After the initial six interviews, some small changes were made to the wording of the categories ('Corporation' for 'Company' in Occupations, and the inclusion of LaTeX with XML in Selection Criteria). An additional 14 participants were recruited, selected on the same basis. The sessions were held in a non-work environment with only this author and the subject present. The pilot sessions were entered on the paper form and transcribed later to a spreadsheet, but the subsequent 14

sessions were recorded directly into a spreadsheet, with the paper form used only as a prompt. Responses were shown to the subject at the end of the interview for their agreement (two forms were corrected owing to a mis-heard product name and a wrong date).

3.1.3 Analysis

The 20 participants were a relatively homogeneous group, with a substantial amount of shared background and experience. All but one of the six independent consultants also worked with a professional institution (and one of those also worked with a corporation); of the remaining 13 corporate affiliates, four were also associated with a university, and three with a professional institution (see Figure 3.1 on the preceding page).

3.1.3.1 Q.1 Background variables

Almost all participants were experienced in Unix (19) and Windows (17), with Apple Mac experience for 13 of them (just over half also had MS-DOS experience), and a few had experience on other systems (Figure 3.2).

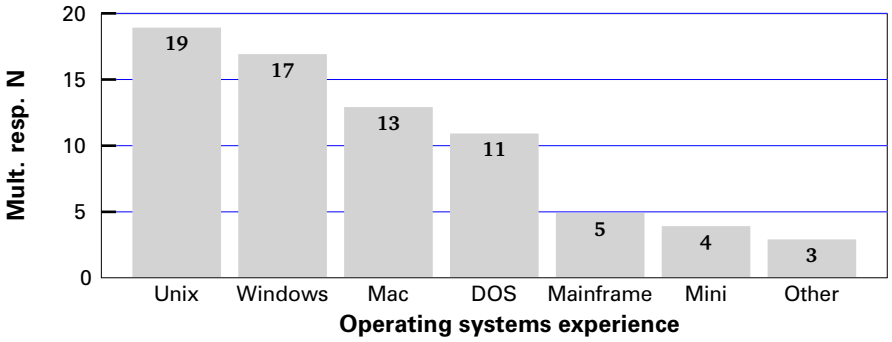


Figure 3.2: Expert survey: Q.1 Background variables: Operating systems experience

Similarly, and by design, almost all had experience with SGML (20) and XML (18), and over half with LaTeX (11); experience with earlier and other systems was negligible (Figure 3.3 on the following page).

The distribution of years of experience showed the majority in the two groups 6–10 and 11–15 years (Figure 3.4 on the next page); with only one with less and one with more.

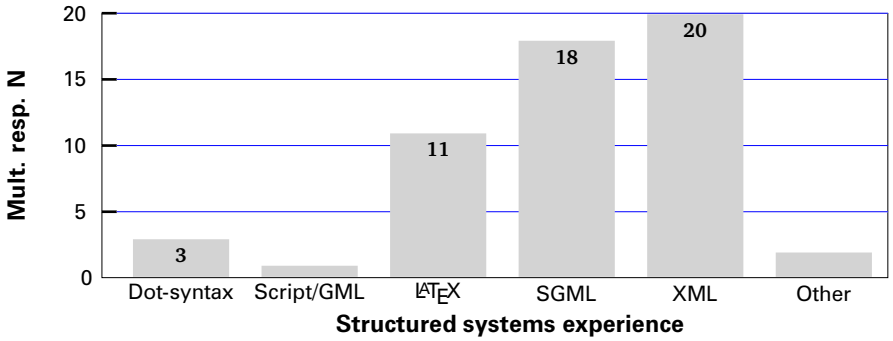


Figure 3.3: Expert survey: Q.1 Background variables: Structured systems experience

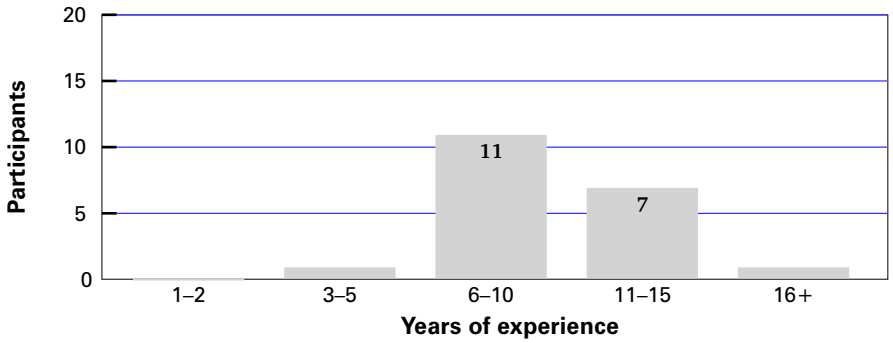


Figure 3.4: Expert survey: Q.1 Background variables: Years of experience

Experience of the major application groups was approximately equal: 18 with experience of document systems and 17 with data systems (Figure 3.5; see section 1.1.3.3 on page 22 for the distinction between 'document' and 'data' systems). The other three categories recorded overlap with these: e-commerce is largely a data application, whereas repositories and web-only systems are both a mixture of document and data. Three different sets each of 10 participants all had experience in these areas as well.

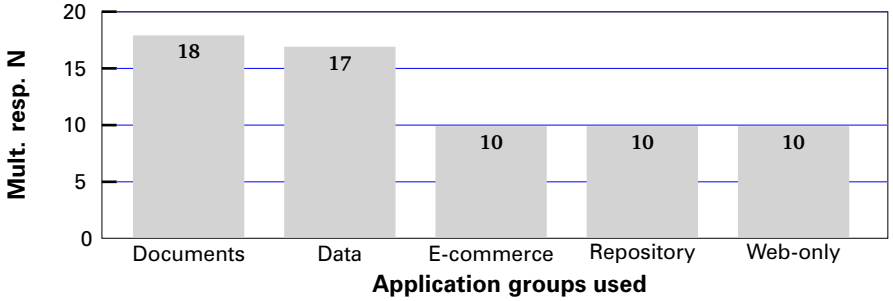


Figure 3.5: Expert survey: Q.1 Background variables: Operating systems experience

3.1.3.2 Q.2 Selection criteria

This question was in two parts. The first asked what general selection criteria the subject used when specifying software for a project; the second asked in more detail exactly what software the subject would choose for their *own* work.

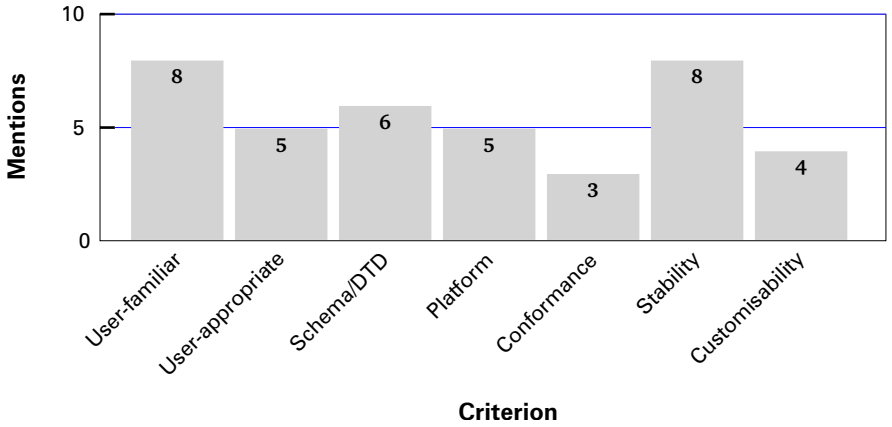


Figure 3.6: Expert survey: Q.2 What criteria do you use when selecting editing software?

3.1.3.2.1 Selection of software for a project This was an open-ended question, so the answers were recoded as multiple responses (Figure 3.6). Stability of the program and familiarity of the interface to the user were the two most often mentioned (eight times each); followed by the requirement to handle a specific DTD or Schema (six mentions).

Familiarity of the interface to the end-user was emphasised by several participants when this criterion was recorded. Because of the complex nature of the markup, and its exposure in the interface, significant training is sometimes required for new users, and the use of an already-familiar interface paradigm is seen as a way to minimise the additional cost of training. As we later found in the Requests Analysis (section 3.3.5 on page 168), customer product requirements sometimes specify that the interface 'must be "*Word*-like"', which tends to support this view.

The requirement for specific DTD/Schema support (6) is important: while almost every editor tested uses a Schema or DTD when specified for a given document type or application, some are limited to those Schemas or DTDs which have been provided in a pre-compiled format by the vendor, or to those which can be compiled afresh with a separate program. This is often the case with security applications or those with a dependency involving risk to life (documentation for medical equipment, transport systems maintenance, and military systems, for

example), where it is essential that only an authorised, centrally-controlled DTD or Schema is used. This may mean that enabling an editor for a new application with a different DTD or Schema might require a significant amount of programming, styling, and testing before the product can be used. Other products, however, can compile a DTD or Schema on-the-fly each time a document is opened, and they appear to perform no check to see if this was the same one as used on a previous occasion of editing the same document.

Availability for a specific platform was mentioned five times, as was the need for the facilities to be appropriate for the task (explained as the avoidance of systems with unnecessary features that would confuse the user or get in the way).

The ability of the software to be customised was mentioned four times, which is perhaps to be expected, given that many of the systems developed by experts at a corporate or institutional level would normally require significant customisation, and that they would naturally seek out software which lent itself to this.

Conformance was mentioned only three times: it is taken for granted that any system proclaiming itself to be XML-conformant or LaTeX-conformant will actually be so. Particularly in the case of XML, 'conformance' is a tightly-specified aspect of the system, being bound as it is to a measurable (in this case, ISO) standard (ISO 8879:1985, 1985).

3.1.3.2.2 Personal choice of editor This part of the question sought to identify the participants' product preferences for a range of 10 common example tasks (see the list on page 108), both for the act of writing the document themselves and for the act of editing someone else's writing. Table 3.1 on the facing page shows a summary of the different products mentioned.

We consider the two modes of use (writing and editing) separately below. The overall responses for both modes are shown in Figure 3.7 on page 116, combining the responses for all 10 application scenarios.

For writing The highest number of mentions were for *Emacs* (38 occurrences) and *XMetaL* (34). The choice of *Emacs* is unsurprising, given the systems background of the expert panel, its status as the programmers' editor of choice for over three decades, and the robust reputation of the *psgml* and *nxml* 'modes'. *XMetaL* is a synchronous typographical editor in widespread use, and its ranking as a close second to *Emacs* merited further scrutiny.

Table 3.1: Editors mentioned in the Expert survey

Editor	Vendor	W	E	Comment
epcEdit	Koch & Halstenberg	3	–	ST editor (DTDs only) with limited GUI stylesheet controls. Supplied as cross-platform Tcl/Tk source.
EPIC	Arbortext	22	29	ST editor with exhaustive styling control, customisation, and mathematics editing. Windows-only now, but the *de facto* standard for heavy-duty technical work.
Emacs	FSF	38	10	FLOSS plaintext editor with full-featured XML and other modes. Extensively programmable, multiplatform, the accepted baseline and fallback.
FrameMaker	Adobe	14	12	DTP system able to work with XML. Obsolescent.
LyX	FSF	3	3	FLOSS multiplatform ST editor for LaTeX.
OpenOffice	Sun and others	11	18	FLOSS multiplatform ST wordprocessor (ODF).
Other	n/a	3	13	This category included a dozen assorted editors, some of which are now obsolete.
SciWord	MacKichan	7	–	ST editor for LaTeX with mathematics editing.
Serna	Syntext	5	4	FLOSS and commercial ST editor with emphasis on DITA.
TextPad	Helios	3	1	Windows-only plaintext editor with some markup support.
Word	Microsoft	12	24	ST wordprocessor (OOXML) with separate XML mode (W3C Schemas only).
WordPerfect	Corel	10	9	ST wordprocessor with separate XML mode (DTDs only).
XED	Henry S. Thompson	3	3	FLOSS plaintext editor (well-formed XML only).
XML Spy	Altova	15	28	XML IDE with extensive support for associated processing technologies.
XMLMind	Pixware	1	3	Multiplatform ST editor with emphasis on DITA and DocBook.
XMLWriter	Wattle	4	8	Windows plaintext GUI editor with formatting preview.
XMetaL	JustSystems	34	10	ST editor concentrating on DITA. One of the earlier XML editors (originally from SoftQuad).
oXygen	SyncRO Soft	5	12	Multiplatform editor/formatter with extensive DITA, DocBook, and TEI support.
vi	Bill Joy and others	7	9	FLOSS plaintext Unix editor with XML support in *Vim*.

[W=writing; E=editing; ST=Synchronous Typographical]

However, *EPIC* scored more consistently than either *Emacs* or *XMetaL*, but its highest individual mention was for writing the ski-lift documentation. Overall, *EPIC* was mentioned more times than any other product, but those mentions were distributed more widely across the categories than was the case for *Emacs* and *XMetaL*.

In Table 3.2 on page 117 we can see (shaded cells) these two latter products selected approximately equally for the first three document types (PDA article, owner manual, and romantic novel), with *Emacs* ahead for the next two types, code documentation and personal memoirs; and *XMetaL* ahead for the later types of children's book and the book on architecture.

However, when the selection criteria in Q.2 are taken into account (Table 3.3 on page 118 and Table 3.4 on page 120), we can see that while *Emacs* was selected much more frequently by participants considering user-familiarity, user-appropriateness, and stability (bearing in mind that the 'user' in this case is

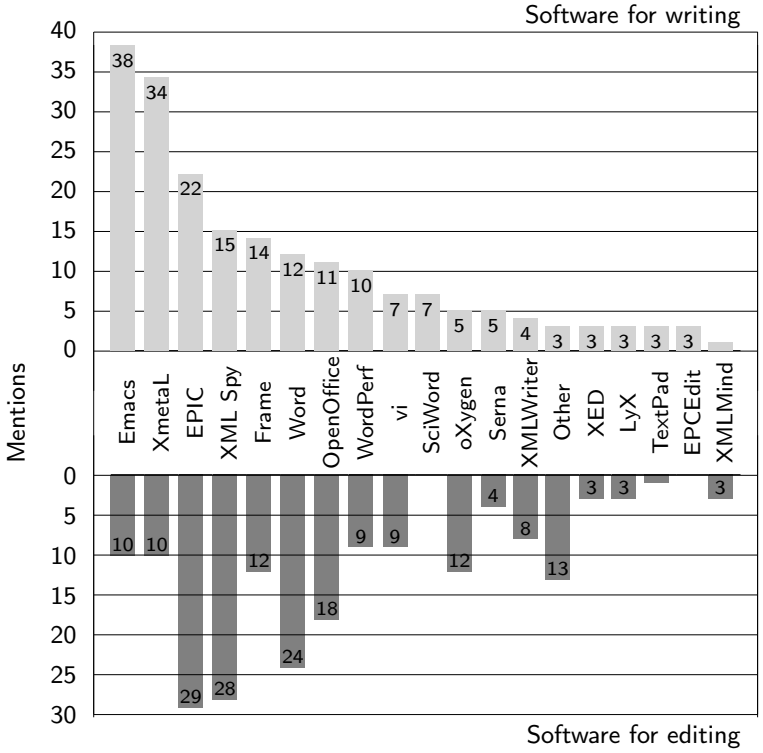


Figure 3.7: Expert survey: Q.2 What would be your personal choice of product [. . .] for a) writing and b) editing?

the expert), *XMetaL* was selected overwhelmingly by the participants who mentioned conformance and customisability.

While both products are conformant and customisable, *Emacs*' conformance depends on the finger-dexterity of the user, as it is a plaintext editor, and inadvertent trespass on the markup characters themselves is an ever-present danger, something that is impossible in a typographically-controlled interface such as *XMetaL*, where only the text content is editable in the character sense. *Emacs* customisability requires learning the eLisp language (a skill generally confined to computer scientists) as well as having a familiarity with the dozens of other 'modes' with which *psgml*, *nxml*, and related software has to cooperate; on the other hand, *XMetaL*'s customisability uses more common scripting languages (*Jscript*, *VBScript*, *Perl*, or *Python*), making it more accessible to the technical user.

The fourth-highest ranking editor overall was *XML Spy* (43 mentions, only one fewer than *XMetaL*), and as with *EPIC*, the mentions were distributed across all document types at the 1–4 level only. In both these cases, this may reflect the products' images as good general-purpose XML workhorses, capable of editing

Table 3.2: Expert survey: Q.2 Personal choice of editor for writing and editing (detail)

	Article on PDAs		Owner Manual		Romantic Novel		Code Documentation		Your Own Memoirs		Ski Lift Maint Manual		Technical Paper		Childrens Book		Oxford Architecture		Consumer Leaflets		*Total Sample*
	Writing	Editing	Writing	Editing	Writing	Editing	Writing	Editing	Writing	Editing	Writing	Editing	Writing	Editing	Writing	Editing	Writing	Editing	Writing	Editing	
Total Sample	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	400 100.0%
EDITOR																					
EPIC	2 10.0%	3 15.0%	3 15.0%	4 20.0%	2 10.0%	2 10.0%	1 5.0%	2 10.0%	1 5.0%	2 10.0%	5 25.0%	4 20.0%	2 10.0%	3 15.0%	2 10.0%	3 15.0%	2 10.0%	3 15.0%	2 10.0%	3 15.0%	51 12.8%
Emacs	4 20.0%	1 5.0%	4 20.0%	1 5.0%	4 20.0%	1 5.0%	6 30.0%	1 5.0%	7 35.0%	1 5.0%	3 15.0%	1 5.0%	4 20.0%	2 10.0%	1 5.0%	- 	2 10.0%	1 5.0%	3 15.0%	1 5.0%	48 12.0%
XMetaL	5 25.0%	1 5.0%	4 20.0%	1 5.0%	6 30.0%	1 5.0%	1 5.0%	1 5.0%	2 10.0%	1 5.0%	2 10.0%	1 5.0%	3 15.0%	1 5.0%	4 20.0%	1 5.0%	4 20.0%	1 5.0%	3 15.0%	1 5.0%	44 11.0%
XML Spy	2 10.0%	4 20.0%	2 10.0%	4 20.0%	2 10.0%	1 5.0%	2 10.0%	2 10.0%	- 	1 5.0%	2 10.0%	4 20.0%	2 10.0%	4 20.0%	- 	2 10.0%	1 5.0%	2 10.0%	2 10.0%	4 20.0%	43 10.8%
Word	- 	2 10.0%	- 	1 5.0%	1 5.0%	4 20.0%	1 5.0%	4 20.0%	1 5.0%	4 20.0%	- 	- 	- 	- 	5 25.0%	5 25.0%	4 20.0%	4 20.0%	- 	- 	36 9.0%
OpenOffice	- 	1 5.0%	- 	1 5.0%	2 10.0%	3 15.0%	- 	3 15.0%	3 15.0%	3 15.0%	- 	1 5.0%	- 	1 5.0%	4 20.0%	2 10.0%	2 10.0%	2 10.0%	- 	1 5.0%	29 7.2%
Frame	- 	2 10.0%	3 15.0%	2 10.0%	- 	1 5.0%	- 	1 5.0%	- 	1 5.0%	2 10.0%	2 10.0%	2 10.0%	1 5.0%	2 10.0%	- 	2 10.0%	- 	3 15.0%	2 10.0%	26 6.5%
WordPerfect	1 5.0%	- 	1 5.0%	- 	1 5.0%	2 10.0%	1 5.0%	1 5.0%	1 5.0%	2 10.0%	1 5.0%	- 	1 5.0%	- 	1 5.0%	2 10.0%	1 5.0%	2 10.0%	1 5.0%	- 	19 4.8%
oXygen	1 5.0%	1 5.0%	- 	1 5.0%	- 	1 5.0%	- 	1 5.0%	- 	1 5.0%	1 5.0%	2 10.0%	1 5.0%	2 10.0%	- 	1 5.0%	1 5.0%	1 5.0%	1 5.0%	1 5.0%	17 4.2%
vi	1 5.0%	1 5.0%	- 	1 5.0%	1 5.0%	1 5.0%	3 15.0%	1 5.0%	2 10.0%	1 5.0%	- 	1 5.0%	- 	1 5.0%	- 	1 5.0%	- 	- 	- 	1 5.0%	16 4.0%
XMLWriter	1 5.0%	1 5.0%	1 5.0%	1 5.0%	- 	1 5.0%	- 	1 5.0%	- 	1 5.0%	1 5.0%	1 5.0%	- 	1 5.0%	- 	- 	- 	- 	1 5.0%	1 5.0%	12 3.0%
Serna	1 5.0%	- 	1 5.0%	1 5.0%	- 	- 	- 	- 	- 	- 	1 5.0%	1 5.0%	1 5.0%	1 5.0%	- 	- 	- 	- 	1 5.0%	1 5.0%	9 2.2%
SciWord	1 5.0%	- 	1 5.0%	- 	- 	- 	- 	- 	- 	- 	1 5.0%	- 	1 5.0%	- 	1 5.0%	- 	1 5.0%	- 	1 5.0%	- 	7 1.8%
XED	1 5.0%	1 5.0%	- 	- 	- 	- 	- 	- 	- 	- 	- 	- 	1 5.0%	1 5.0%	- 	- 	- 	- 	1 5.0%	1 5.0%	6 1.5%
LyX	- 	1 5.0%	- 	1 5.0%	1 5.0%	- 	1 5.0%	1 5.0%	1 5.0%	- 	- 	- 	- 	- 	- 	- 	- 	- 	- 	- 	6 1.5%
TextPad	- 	- 	- 	- 	- 	- 	2 10.0%	- 	1 5.0%	- 	- 	- 	- 	- 	- 	- 	- 	- 	- 	1 5.0%	4 1.0%
XMLMind	- 	- 	- 	- 	- 	- 	1 5.0%	- 	- 	- 	- 	1 5.0%	- 	1 5.0%	- 	- 	- 	- 	- 	1 5.0%	4 1.0%
EPCEdit	- 	- 	- 	- 	- 	- 	- 	- 	- 	- 	1 5.0%	- 	1 5.0%	- 	- 	- 	- 	- 	1 5.0%	- 	3 0.8%
Other	- 	1 5.0%	- 	1 5.0%	- 	1 5.0%	1 5.0%	1 5.0%	1 5.0%	1 5.0%	- 	1 5.0%	1 5.0%	1 5.0%	- 	2 10.0%	- 	3 15.0%	- 	1 5.0%	16 4.0%
Missing	- 	- 	- 	- 	- 	1 5.0%	- 	- 	- 	1 5.0%	- 	- 	- 	- 	- 	1 5.0%	- 	1 5.0%	- 	- 	4 1.0%
Base	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	19 95.0%	20 100.0%	20 100.0%	20 100.0%	19 95.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	20 100.0%	19 95.0%	20 100.0%	19 95.0%	20 100.0%	20 100.0%	396 99.0%

Frequency of mention ranked by total, with column percents. Shaded cells show frequencies of 4 and over.

Table 3.3: Expert survey: personal choice of editor for writing (Q.2b) according to software selection criteria (Q.2a)

Editor for writing	Familiar	Appro-priate	Schema/ DTD	Platform	Confor-mance	Stability	Customi-sability	Total Sample
XMetaL	14 20.6% 17.5%	8 11.8% 16.0%	8 11.8% 13.3%	8 11.8% 16.0%	12 17.6% 40.0%	10 14.7% 12.5%	8 11.8% 20.0%	68 100.0% 17.4%
Emacs	22 34.9% 27.5%	10 15.9% 20.0%	9 14.3% 15.0%	7 11.1% 14.0%	– 0.0% 0.0%	15 23.8% 18.8%	– 0.0% 0.0%	63 100.0% 16.2%
EPIC	12 31.6% 15.0%	8 21.1% 16.0%	– 0.0% 0.0%	6 15.8% 12.0%	1 2.6% 3.3%	10 26.3% 12.5%	1 2.6% 2.5%	38 100.0% 9.7%
XML Spy	– 0.0% 0.0%	8 21.1% 16.0%	10 26.3% 16.7%	7 18.4% 14.0%	– 0.0% 0.0%	5 13.2% 6.2%	8 21.1% 20.0%	38 100.0% 9.7%
WordPerfect	– 0.0% 0.0%	10 33.3% 20.0%	10 33.3% 16.7%	– 0.0% 0.0%	– 0.0% 0.0%	10 33.3% 12.5%	– 0.0% 0.0%	30 100.0% 7.7%
Frame	6 20.7% 7.5%	2 6.9% 4.0%	3 10.3% 5.0%	7 24.1% 14.0%	1 3.4% 3.3%	8 27.6% 10.0%	2 6.9% 5.0%	29 100.0% 7.4%
OpenOffice	3 13.0% 3.8%	1 4.3% 2.0%	2 8.7% 3.3%	7 30.4% 14.0%	– 0.0% 0.0%	6 26.1% 7.5%	4 17.4% 10.0%	23 100.0% 5.9%
SciWord	7 33.3% 8.8%	– 0.0% 0.0%	7 33.3% 11.7%	– 0.0% 0.0%	– 0.0% 0.0%	7 33.3% 8.8%	– 0.0% 0.0%	21 100.0% 5.4%
Word	5 27.8% 6.2%	– 0.0% 0.0%	2 11.1% 3.3%	2 11.1% 4.0%	5 27.8% 16.7%	2 11.1% 2.5%	2 11.1% 5.0%	18 100.0% 4.6%
vi	1 8.3% 1.2%	– 0.0% 0.0%	– 0.0% 0.0%	5 41.7% 10.0%	2 16.7% 6.7%	4 33.3% 5.0%	– 0.0% 0.0%	12 100.0% 3.1%
oXygen	5 50.0% 6.2%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	5 50.0% 12.5%	10 100.0% 2.6%
XMLWriter	– 0.0% 0.0%	– 0.0% 0.0%	4 50.0% 6.7%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	4 50.0% 10.0%	8 100.0% 2.1%
TextPad	– 0.0% 0.0%	2 28.6% 4.0%	– 0.0% 0.0%	– 0.0% 0.0%	1 14.3% 3.3%	3 42.9% 3.8%	1 14.3% 2.5%	7 100.0% 1.8%
LyX	– 0.0% 0.0%	– 0.0% 0.0%	3 50.0% 5.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	3 50.0% 7.5%	6 100.0% 1.5%
Serna	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	5 100.0% 16.7%	– 0.0% 0.0%	– 0.0% 0.0%	5 100.0% 1.3%
epcEdit	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	3 100.0% 10.0%	– 0.0% 0.0%	– 0.0% 0.0%	3 100.0% 0.8%
XED	3 100.0% 3.8%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	3 100.0% 0.8%
XMLMind	1 50.0% 1.2%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	1 50.0% 2.5%	2 100.0% 0.5%
Other[a]	1 16.7% 1.2%	1 16.7% 2.0%	2 33.3% 3.3%	1 16.7% 2.0%	– 0.0% 0.0%	– 0.0% 0.0%	1 16.7% 2.5%	6 100.0% 1.5%
Total Sample	80 20.5%	50 12.8%	60 15.4%	50 12.8%	30 7.7%	80 20.5%	40 10.3%	390 100.0%

a. The 'Other' category contained singleton mentions of specialist or unknown systems.

Multiple responses for all document types by multiple responses to selection criteria, ranked by overall frequency of mention, with row and column percents. Shading shows frequencies of 10 and over.

almost any document type; whereas *XMetaL* has a reputation in the technical documentation field with the Organization for the Advancement of Structured Information Standards (OASIS)-defined standard Darwin Information Typing Architecture (DITA) and DocBook formats, and *Emacs* retains its quasi-anomalous status as a multi-purpose editor that happens to have strong XML and LaTeX support.

Microsoft *Word* appears in Table 3.2 on page 117 in fifth position, largely because of the number of mentions it garnered for the children's book and architecture book document types (five and four respectively, both for writing and editing). In the case of children's books, there may be a perception that they are too simple to warrant the expenditure of time and effort in setting up an XML workflow; and that in the case of the book on architecture, that the volume of graphics might outweigh any textual considerations. To this author's knowledge, there have been no published investigations into the choice of software for either type of document.

In sixth and seventh position overall were *OpenOffice* and *FrameMaker* respectively.

3.1.3.2.2.1 For editing The experts' choice was radically different. *Emacs* and *XMetaL* were only mentioned 10 times each, whereas *EPIC* and *XML Spy* were almost level with 29 and 28 mentions respectively, and Microsoft *Word* was a close third with 24 (Figure 3.7 on page 116).

This may be connected with the nature of the task. Editing another writer's structured text involves judgement tasks about what the author intended, and whether the structures used can reliably represent the author's intent. Any change may involve considerable editing of the markup, and the experts' use of *de facto* industrial 'standards' like *EPIC* and *XML Spy* may outweigh the lesser-used *Emacs* and *XMetaL*. The presence of *Word* possibly results from its dominant status rather than any inherent feature.

This is to some extent borne out by the comparison with the experts' selection criteria in Table 3.4 on the following page. *EPIC* rates very strongly with participants who look for stability (15), interface familiarity (14), conformance and customisability (10 each). *XML Spy* rated even higher among those who look for stability (17), but differed substantially from *EPIC* elsewhere, rating higher among those who sought an appropriate interface (12), ability to handle specific

Table 3.4: Expert survey: personal choice of editor for editing (Q.2b) according to software selection criteria (Q.2a)

Editor for editing	Familiar	Appro-priate	Schema/ DTD	Platform	Confor-mance	Stability	Customi-sability	Total Sample
EPIC	14 23.0% 17.5%	5 8.2% 10.0%	– 0.0% 0.0%	7 11.5% 14.0%	10 16.4% 38.5%	15 24.6% 18.8%	10 16.4% 25.0%	61 100.0% 15.8%
XML Spy	– 0.0% 0.0%	12 20.0% 24.0%	10 16.7% 16.7%	10 16.7% 20.0%	6 10.0% 23.1%	17 28.3% 21.2%	5 8.3% 12.5%	60 100.0% 15.5%
Word	1 2.2% 1.2%	15 32.6% 30.0%	5 10.9% 8.3%	2 4.3% 4.0%	6 13.0% 23.1%	12 26.1% 15.0%	5 10.9% 12.5%	46 100.0% 11.9%
OpenOffice	8 22.2% 10.0%	– 0.0% 0.0%	10 27.8% 16.7%	13 36.1% 26.0%	– 0.0% 0.0%	5 13.9% 6.2%	– 0.0% 0.0%	36 100.0% 9.3%
vi	– 0.0% 0.0%	9 33.3% 18.0%	9 33.3% 15.0%	9 33.3% 18.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	27 100.0% 7.0%
WordPerfect	4 14.8% 5.0%	5 18.5% 10.0%	9 33.3% 15.0%	– 0.0% 0.0%	– 0.0% 0.0%	9 33.3% 11.2%	– 0.0% 0.0%	27 100.0% 7.0%
Frame	4 16.7% 5.0%	– 0.0% 0.0%	– 0.0% 0.0%	8 33.3% 16.0%	– 0.0% 0.0%	12 50.0% 15.0%	– 0.0% 0.0%	24 100.0% 6.2%
oXygen	9 37.5% 11.2%	3 12.5% 6.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	3 12.5% 3.8%	9 37.5% 22.5%	24 100.0% 6.2%
XMLWriter	– 0.0% 0.0%	– 0.0% 0.0%	8 50.0% 13.3%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	8 50.0% 20.0%	16 100.0% 4.1%
Emacs	10 90.9% 12.5%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	1 9.1% 1.2%	– 0.0% 0.0%	11 100.0% 2.8%
XMetaL	10 100.0% 12.5%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	10 100.0% 2.6%
LyX	3 33.3% 3.8%	– 0.0% 0.0%	3 33.3% 5.0%	– 0.0% 0.0%	– 0.0% 0.0%	3 33.3% 3.8%	– 0.0% 0.0%	9 100.0% 2.3%
XMLMind	3 33.3% 3.8%	– 0.0% 0.0%	3 33.3% 5.0%	– 0.0% 0.0%	– 0.0% 0.0%	3 33.3% 3.8%	– 0.0% 0.0%	9 100.0% 2.3%
Serna	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	4 100.0% 15.4%	– 0.0% 0.0%	– 0.0% 0.0%	4 100.0% 1.0%
XED	3 100.0% 3.8%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	3 100.0% 0.8%
TextPad	1 50.0% 1.2%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	– 0.0% 0.0%	1 50.0% 2.5%	2 100.0% 0.5%
Other[a]	10 58.8% 12.5%	1 5.9% 2.0%	3 17.6% 5.0%	1 5.9% 2.0%	– 0.0% 0.0%	– 0.0% 0.0%	2 11.8% 5.0%	17 100.0% 4.4%
Total Sample	80 20.7%	50 13.0%	60 15.5%	50 13.0%	26 6.7%	80 20.7%	40 10.4%	386 100.0%

a. The 'Other' category contained singleton mentions of specialist or unknown systems.

Multiple responses for all document types by multiple responses to selection criteria, ranked by overall frequency of mention, with row and column percents. Shaded cells show frequencies of 10 and over.

DTDs or Schemas (10), and cross-platform availability (10).

Word also scored highly among those seeking an appropriate interface (15) and stability (12), whereas *OpenOffice* rated higher among those looking for cross-platform availability (13) and ability to handle specific DTDs or Schemas (10). The only other scores at this level were *FrameMaker*, which rated 12 under stability, and *Emacs* and *XMetaL*, which both rated 10 under interface familiarity.

The only other editor mentioned more than half the number of times of *Word*'s score in Figure 3.7 on page 116 was *OpenOffice* (18). The relatively high score for the 'Other' category may reflect some experts' use of little-known or specialist software in which they may feel particular confidence.

3.1.3.2.3 Analysis When we look at the division between types of document (Table 3.2 on page 117), we can see that while *EPIC* scored highly, the mentions were distributed across most document types in twos and threes, with only two frequencies of four (owner manual and ski-lift maintenance documents). *XML Spy*, on the other hand, the other top ranker for editing, scored four mentions no less than five times, with other scores at the one and two levels. Ones and twos are also very evident in the scores for *Emacs* and *XMetaL*, which had ranked so highly for the experts' own writing, whereas Microsoft *Word* had four mentions for four document types, and five for one of them (the children's book).

We can make a tentative conclusion from the foregoing that these results may indicate a substantial difference in the requirements of software for writing original work compared with editing existing work. This is an important distinction when it comes to judging the provisions of an editing system, because the person who habitually edits another's work is considered more likely to be experienced in using the tools for doing so, whereas no such guarantee can be adduced in the case of a writer. This distinction will be important when we come to look at the features offered by the software.

It is also important to note that the software market and the products available now change very rapidly in this field, although some have become permanent fixtures. This part of the study was conducted in 2004–5, and there are now other editors available, and some have fallen by the wayside. However, the critical fact is that the distinction can be made between different *types* of editor, rather than between different makes or brands, and we shall be making us of this fact in the Software Analysis in section 3.2 on page 133.

3.1.3.3 Q.3 Products not used

Among the experts were a few who felt unable to use specific products because of commercial or technical restrictions or limitations. The high price of some XML software is alluded to later (section 3.2.1 on page 134), and there are are concerns about platform dependence which may affect users with a wide variety of operating systems. The numbers, however, were very small (Figure 3.8), and are unlikely to have any major effect.

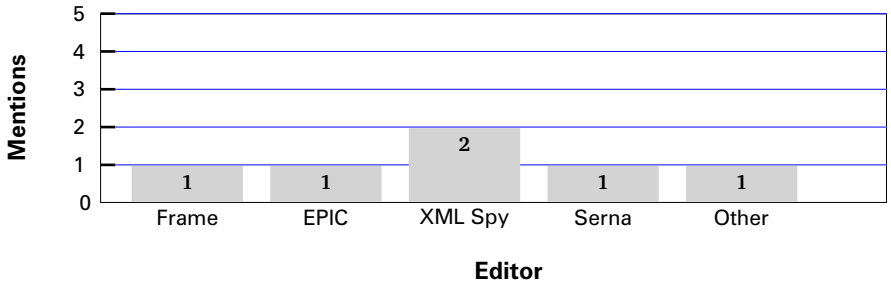


Figure 3.8: Expert survey: Q.3 Are there products you would prefer to use but cannot (for reasons of price, licensing, availability, platforms, etc)?

3.1.3.4 Q.4 Specially useful facilities

Devotees of specific software (in all fields, not just XML or LaTeX) are well-known anecdotally for their proselytism — the 'war' between *vi* and *Emacs* users is epic material.

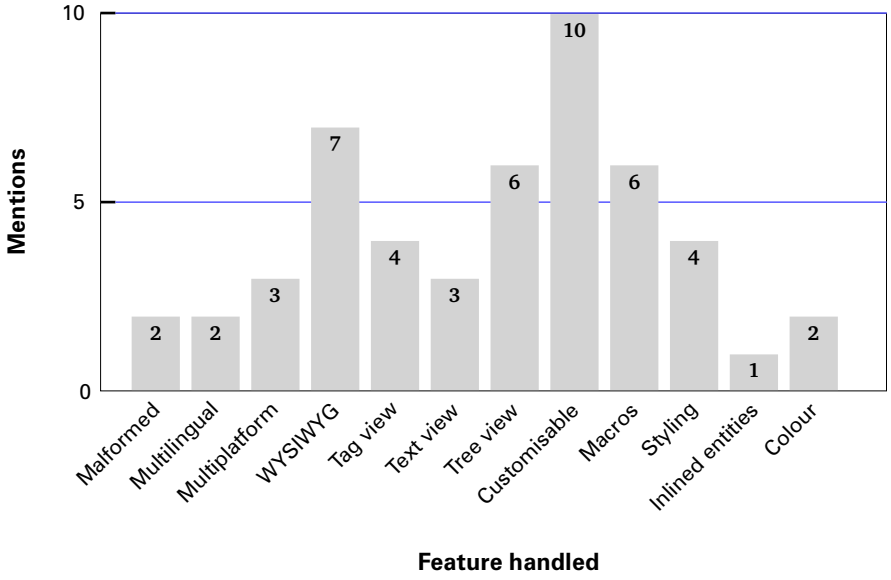


Figure 3.9: Expert survey: Q.4 Can you identify up to three things your preferred software does which you consider marks it out as specially useful?

This question tried to detect if one or more specific features were sufficiently important to make a product an 'editor of choice' (Figure 3.9 on the preceding page). Customisability was the facility most often mentioned (10). Given the participants' nature as experts and consultants, this aspect is perhaps unsurprising, as a large part of their work would involve customising software for clients or colleagues, but it may partly explain the predilection for software with a reputation for customisability seen in section 3.1.3.2 on page 113.

However, there is a conflict between this ranking of customisability and its application when we consider the criteria for editor selection (Table 3.5 on the following page), where a tendency to regard customisability as a selection criterion is not apparent among those regarding customisability as a useful feature of their own editor choice. This may be because the expert's ability to customise an application is a different requirement to what software they would recommend to a user.

Second ranked was WYSIWYG (7). We have treated elsewhere of the nature of this phenomenon (section 1.1.7 on page 39), but here and elsewhere in this study the term was used both as a shorthand for the synchronous typographical interface also defined in section 1.1.7 on page 39 and for the graphical claims of the software vendors. It was clear during the interviews that many of the participants visually mimicked quotations marks around 'WYSIWYG' when using the term, or used other facial expressions to indicate a grudging acceptance of the fact that what the user sees is not really what she gets, only an approximation.

Third equal at six mentions each are the provision of a tree view of the document structure, and the availability of macros. The conflicting views of users on tree displays are explained in section 1.1.3.2 on page 20 and in 'Structure-view editors' in the list below (page 172), but it is clearly seen to be a significant advantage to software which provides them (specific products with this feature are shown in section 3.2 on page 133). Similarly with macros: users who are prepared to give time to learn a macro or scripting language may be more enthusiastic about them because of the time they have invested. This is particularly true of LaTeX editors, as the whole TeX system is based on macros.

Other facilities mentioned included a 'tag view' (4), which is the ability to display a typographical rendering of the document with the markup symbolically represented, without actually going as far as the 'bare-metal' level of a plaintext display; the availability of styling (4), which is the ability to create or modify a stylesheet for consistent application of styles while actually editing the document

Table 3.5: Expert survey: Useful features of an editor (Q.4) according to software selection criteria (Q.2a)

Useful	Familiar	Appro-priate	Schema/ DTD	Platform	Confor-mance	Stability	Customi-sability	Total Sample
Customisable	4	4	2	4	–	4	2	20
	20.0%	20.0%	10.0%	20.0%	0.0%	20.0%	10.0%	100.0%
	19.0%	28.6%	14.3%	30.8%	0.0%	21.1%	20.0%	20.4%
WYSIWYG	4	2	3	1	–	4	2	16
	25.0%	12.5%	18.8%	6.2%	0.0%	25.0%	12.5%	100.0%
	19.0%	14.3%	21.4%	7.7%	0.0%	21.1%	20.0%	16.3%
Macros	3	3	2	2	–	2	–	12
	25.0%	25.0%	16.7%	16.7%	0.0%	16.7%	0.0%	100.0%
	14.3%	21.4%	14.3%	15.4%	0.0%	10.5%	0.0%	12.2%
Tree view	1	1	1	1	3	3	2	12
	8.3%	8.3%	8.3%	8.3%	25.0%	25.0%	16.7%	100.0%
	4.8%	7.1%	7.1%	7.7%	42.9%	15.8%	20.0%	12.2%
Styling	1	2	1	2	1	2	1	10
	10.0%	20.0%	10.0%	20.0%	10.0%	20.0%	10.0%	100.0%
	4.8%	14.3%	7.1%	15.4%	14.3%	10.5%	10.0%	10.2%
Tag view	1	1	1	–	1	1	1	6
	16.7%	16.7%	16.7%	0.0%	16.7%	16.7%	16.7%	100.0%
	4.8%	7.1%	7.1%	0.0%	14.3%	5.3%	10.0%	6.1%
Text view	–	1	–	–	2	2	1	6
	0.0%	16.7%	0.0%	0.0%	33.3%	33.3%	16.7%	100.0%
	0.0%	7.1%	0.0%	0.0%	28.6%	10.5%	10.0%	6.1%
Multiplatform	2	–	1	1	–	–	1	5
	40.0%	0.0%	20.0%	20.0%	0.0%	0.0%	20.0%	100.0%
	9.5%	0.0%	7.1%	7.7%	0.0%	0.0%	10.0%	5.1%
Multilingual	2	–	1	–	–	1	–	4
	50.0%	0.0%	25.0%	0.0%	0.0%	25.0%	0.0%	100.0%
	9.5%	0.0%	7.1%	0.0%	0.0%	5.3%	0.0%	4.1%
Colour	1	–	1	1	–	–	–	3
	33.3%	0.0%	33.3%	33.3%	0.0%	0.0%	0.0%	100.0%
	4.8%	0.0%	7.1%	7.7%	0.0%	0.0%	0.0%	3.1%
Malformed	1	–	1	1	–	–	–	3
	33.3%	0.0%	33.3%	33.3%	0.0%	0.0%	0.0%	100.0%
	4.8%	0.0%	7.1%	7.7%	0.0%	0.0%	0.0%	3.1%
Inlined ents	1	–	–	–	–	–	–	1
	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%
	4.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.0%
Total Sample	21	14	14	13	7	19	10	98
	21.4%	14.3%	14.3%	13.3%	7.1%	19.4%	10.2%	100.0%

Multiple responses for all useful features by multiple responses to selection criteria, ranked by overall frequency of mention, with row and column percents. Shaded cells show frequencies of 4 and over.

(many systems explicitly do not provide this, as organisations with strict 'house' requirements for styling do not want writers and editors making unauthorised modifications).

At a lower level of mention were multiplatform availability (3) and a plaintext view of the document (2); the ability to handle malformed documents (2), multilingual capability (2), and the colour-coding of markup ('colourisation' or 'syntactic fontification' in some domains); and the ability to handle inlined entities (the inclusion at the paragraph content level of text fragments sometimes stored in external files). Malformed documents are sometimes a special concern for experts, as they pose special problems which require a very deep

understanding of the markup, and what can go wrong with it. A persuasive argument for this feature has been well made on several occasions (Birnbaum, 1997; Birnbaum & Mundie, 1999; Laforest & Flory, 2002).

3.1.3.5 Q.5 Particularly poor facilities

This is, in effect, the reverse of the preceding question (on especially good facilities), and it produced a smaller group of responses (Figure 3.10). The most frequent mention (6) was the inability of some editors to support the whole of the Cascading Style Sheets (CSS) specification (Bos, Lie, Lilley, & Jacobs, 2008)). Not all editors use CSS — some use XML-based stylesheets like XSLT, or proprietary systems not accessible to the user — but CSS has become an indispensable tool for web publishing as well as paper publishing, and those editors supporting it are expected to do so fully.

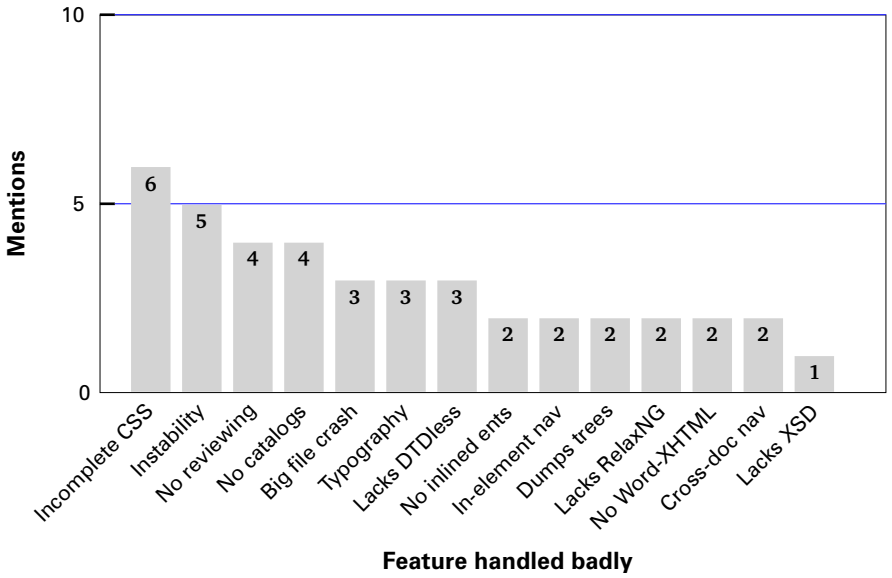


Figure 3.10: Expert survey: Q.5 Can you identify up to three things about any of the structured-document software that you have tested or used which you consider are particularly poor?

Instability was second highest (5), followed by the lack of a facility to allow review comments and changes by subsequent editors (4) and a lack of support for XML Catalogs (4), which are a standard mechanism for the identification of DTDs and other document components held in external files — again, a vital facility for publishing.

Others mentioned were crashing on big files (as distinct from general instability: 3); poor typography (3); the lack of a DTDless mode (3; see section 1.1.4.1 on

Table 3.6: Expert survey: Poor features of an editor (Q.5) according to software selection criteria (Q.2a)

Poor features	Familiar	Appro-priate	Schema/ DTD	Platform	Confor-mance	Stability	Customi-sability	Total Sample
Instability	1	1	2	2	1	2	3	12
	8.3%	8.3%	16.7%	16.7%	8.3%	16.7%	25.0%	100.0%
	5.6%	12.5%	16.7%	28.6%	14.3%	14.3%	27.3%	15.6%
Incomplete CSS	2	2	2	–	2	1	2	11
	18.2%	18.2%	18.2%	0.0%	18.2%	9.1%	18.2%	100.0%
	11.1%	25.0%	16.7%	0.0%	28.6%	7.1%	18.2%	14.3%
No catalogs	1	1	1	3	–	2	–	8
	12.5%	12.5%	12.5%	37.5%	0.0%	25.0%	0.0%	100.0%
	5.6%	12.5%	8.3%	42.9%	0.0%	14.3%	0.0%	10.4%
No reviewing	3	–	2	1	–	2	–	8
	37.5%	0.0%	25.0%	12.5%	0.0%	25.0%	0.0%	100.0%
	16.7%	0.0%	16.7%	14.3%	0.0%	14.3%	0.0%	10.4%
Typography	–	1	1	1	1	2	2	8
	0.0%	12.5%	12.5%	12.5%	12.5%	25.0%	25.0%	100.0%
	0.0%	12.5%	8.3%	14.3%	14.3%	14.3%	18.2%	10.4%
Lacks DTDless	1	1	2	–	–	2	1	7
	14.3%	14.3%	28.6%	0.0%	0.0%	28.6%	14.3%	100.0%
	5.6%	12.5%	16.7%	0.0%	0.0%	14.3%	9.1%	9.1%
Big file crash	2	–	1	–	–	–	1	4
	50.0%	0.0%	25.0%	0.0%	0.0%	0.0%	25.0%	100.0%
	11.1%	0.0%	8.3%	0.0%	0.0%	0.0%	9.1%	5.2%
Cross.doc nav	–	1	1	–	1	1	–	4
	0.0%	25.0%	25.0%	0.0%	25.0%	25.0%	0.0%	100.0%
	0.0%	12.5%	8.3%	0.0%	14.3%	7.1%	0.0%	5.2%
In.element nav	1	1	–	–	–	1	–	3
	33.3%	33.3%	0.0%	0.0%	0.0%	33.3%	0.0%	100.0%
	5.6%	12.5%	0.0%	0.0%	0.0%	7.1%	0.0%	3.9%
Lacks RelaxNG	1	–	–	–	1	–	1	3
	33.3%	0.0%	0.0%	0.0%	33.3%	0.0%	33.3%	100.0%
	5.6%	0.0%	0.0%	0.0%	14.3%	0.0%	9.1%	3.9%
No inlined ents	2	–	–	–	–	–	1	3
	66.7%	0.0%	0.0%	0.0%	0.0%	0.0%	33.3%	100.0%
	11.1%	0.0%	0.0%	0.0%	0.0%	0.0%	9.1%	3.9%
No Word.XHTML	1	–	–	–	1	1	–	3
	33.3%	0.0%	0.0%	0.0%	33.3%	33.3%	0.0%	100.0%
	5.6%	0.0%	0.0%	0.0%	14.3%	7.1%	0.0%	3.9%
Dumps trees	2	–	–	–	–	–	–	2
	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%
	11.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.6%
Lacks XSD	1	–	–	–	–	–	–	1
	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%
	5.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.3%
Total Sample	18	8	12	7	7	14	11	77
	23.4%	10.4%	15.6%	9.1%	9.1%	18.2%	14.3%	100.0%

Multiple responses for all poor features by multiple responses to selection criteria, ranked by overall frequency of mention, with row and column percents.

page 31); the lack of inline entities (2; see the comment in section 3.1.3.4 on page 125); a lack of navigation facilities between element types (2); a tendency to crash and 'dump trees'[1] (2); an inability to handle RelaxNG schemas (2; see section 1.1.4.1 on page 27); lack of ability to import from *Word* to XHTML (2); no facility for navigation between separate documents (2); and a lack of support for W3C Schemas (1).

3.1.3.6 Q.6 Facilities lacking

The three features identified here (see Figure 3.11 and Table 3.7 on the following page) were support for XML Schema Definition (XSD) files [W3C Schemas] (3), the ability to handle typographic formatting properly (3), and the provision of facilities for handling the conversion between *Word* files and XHTML (3).

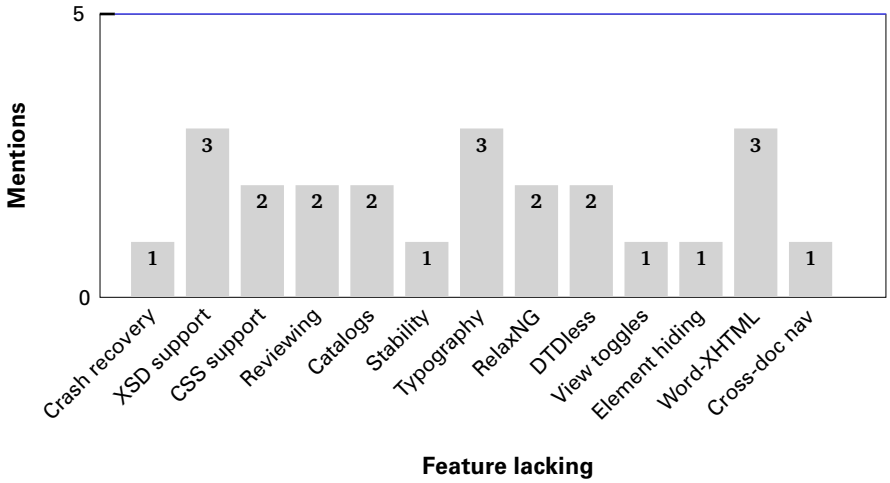


Figure 3.11: Expert survey: Q.6 Are there any other facilities in which structured-document software you have experience of is specially lacking?

Below this, with just two mentions each, are support for CSS styling; for reviewing (the ability to highlight or signal text that has been changed during the editorial process); the use of XML Catalogs (but see section 3.1.3.7 on page 130); the use of RelaxNG schema control; and the use of DTDless documents. These features would typically not be found in some low-end editing systems.

[1] To spew out the tree-structure of the document as a long series of program locations, enabling the programmer to trace backwards through the code to locate the error.

Table 3.7: Expert survey: Features lacking in an editor (Q.6) according to software selection criteria (Q.2a)

Lack	Familiar	Appro-priate	Schema/ DTD	Platform	Confor-mance	Stability	Customi-sability	Total Sample
Typography	1	2	3	–	–	2	1	9
	11.1%	22.2%	33.3%	0.0%	0.0%	22.2%	11.1%	100.0%
	11.1%	33.3%	42.9%	0.0%	0.0%	20.0%	20.0%	19.1%
Word.XHTML	1	2	1	2	–	1	–	7
	14.3%	28.6%	14.3%	28.6%	0.0%	14.3%	0.0%	100.0%
	11.1%	33.3%	14.3%	28.6%	0.0%	10.0%	0.0%	14.9%
Catalogs	–	–	1	1	–	1	1	4
	0.0%	0.0%	25.0%	25.0%	0.0%	25.0%	25.0%	100.0%
	0.0%	0.0%	14.3%	14.3%	0.0%	10.0%	20.0%	8.5%
CSS support	–	1	1	–	1	–	1	4
	0.0%	25.0%	25.0%	0.0%	25.0%	0.0%	25.0%	100.0%
	0.0%	16.7%	14.3%	0.0%	33.3%	0.0%	20.0%	8.5%
Reviewing	1	–	–	1	–	2	–	4
	25.0%	0.0%	0.0%	25.0%	0.0%	50.0%	0.0%	100.0%
	11.1%	0.0%	0.0%	14.3%	0.0%	20.0%	0.0%	8.5%
XSD support	1	–	–	1	1	1	–	4
	25.0%	0.0%	0.0%	25.0%	25.0%	25.0%	0.0%	100.0%
	11.1%	0.0%	0.0%	14.3%	33.3%	10.0%	0.0%	8.5%
Cross.doc nav	–	–	–	–	1	1	1	3
	0.0%	0.0%	0.0%	0.0%	33.3%	33.3%	33.3%	100.0%
	0.0%	0.0%	0.0%	0.0%	33.3%	10.0%	20.0%	6.4%
DTDless	1	1	–	–	–	1	–	3
	33.3%	33.3%	0.0%	0.0%	0.0%	33.3%	0.0%	100.0%
	11.1%	16.7%	0.0%	0.0%	0.0%	10.0%	0.0%	6.4%
RelaxNG	1	–	1	1	–	–	–	3
	33.3%	0.0%	33.3%	33.3%	0.0%	0.0%	0.0%	100.0%
	11.1%	0.0%	14.3%	14.3%	0.0%	0.0%	0.0%	6.4%
Crash recovery	1	–	–	–	–	–	1	2
	50.0%	0.0%	0.0%	0.0%	0.0%	0.0%	50.0%	100.0%
	11.1%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	4.3%
Stability	–	–	–	1	–	1	–	2
	0.0%	0.0%	0.0%	50.0%	0.0%	50.0%	0.0%	100.0%
	0.0%	0.0%	0.0%	14.3%	0.0%	10.0%	0.0%	4.3%
Element hiding	1	–	–	–	–	–	–	1
	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%
	11.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.1%
View toggles	1	–	–	–	–	–	–	1
	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%
	11.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.1%
Total Sample	9	6	7	7	3	10	5	47
	19.1%	12.8%	14.9%	14.9%	6.4%	21.3%	10.6%	100.0%

Multiple responses for all features lacking by multiple responses to selection criteria, ranked by overall frequency of mention, with row and column percents. Shaded cells show frequencies of 10 and over.

3.1.3.7 Q.7 Failure to identify

This question dealt with behaviour or performance by the individual, rather than by the software, although from the point of view of usability, the failure of the program to provide the necessary affordances is clearly a factor: if an expert cannot work out how to do a task, then there is little hope for the average user.

The three top categories represent three entirely different aspects of editing software (see Figure 3.12 and Table 3.8 on the next page).

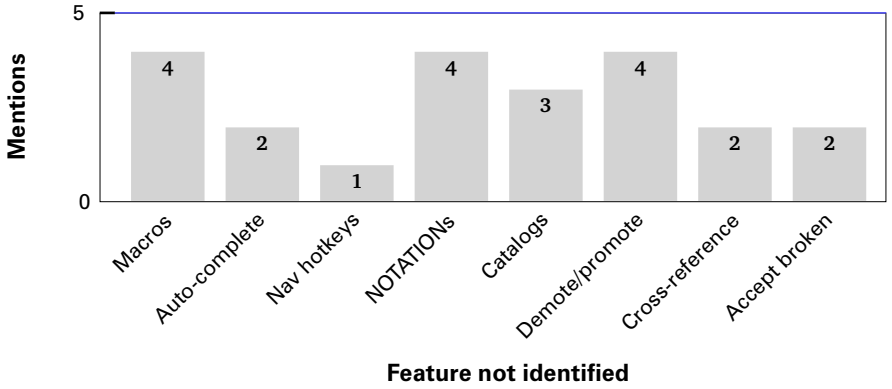


Figure 3.12: Expert survey: Q.7 While using a structured-document software product, have you ever failed to identify how to do something the product was actually capable of?

- Macros (4) are programmable extensions of an editor's behaviour which are optional add-ons rather than inbuilt features. They require a significant level of programming expertise to create, and anecdotally they are poorly documented.
- NOTATIONs (4) are a feature of the XML language inherited from SGML which enable a document to make reference to an external file which is not in XML format, such as an image, video, or sound file. A program like an editor which cannot handle such references cannot be said to be conformant to the XML specification.
- The promotion and demotion of blocks of text (4) is a key requirement of editing systems. It allows the writer to make a section into a subsection, or to make a sub-list into a top-level list, or *vice versa*, and to handle the markup changes involved as well as the typographic requirements, numbering, and any cross-references that may occur.

The handling of Catalogs (3) is an optional feature. The XML Catalog Specification (Walsh, 2001) is a method for resolving the external entities referenced in an XML

document, and is heavily used in publishing. Even with the requirement in XML for the use of Universal Resource Indicators (URI)s as SYSTEM identifiers, the dereferencing of external references remains problematic in some cases.

Table 3.8: Expert survey: Failure to identify abilities in an editor (Q.7) according to software selection criteria (Q.2a)

Failure	Familiar	Appro-priate	Schema/ DTD	Platform	Confor-mance	Stability	Customi-sability	Total Sample
Demote/promote	2	1	2	1	–	2	2	10
	20.0%	10.0%	20.0%	10.0%	0.0%	20.0%	20.0%	100.0%
	25.0%	16.7%	22.2%	20.0%	0.0%	22.2%	40.0%	22.2%
NOTATIONs	1	–	3	1	1	2	2	10
	10.0%	0.0%	30.0%	10.0%	10.0%	20.0%	20.0%	100.0%
	12.5%	0.0%	33.3%	20.0%	33.3%	22.2%	40.0%	22.2%
Macros	2	1	1	2	–	2	–	8
	25.0%	12.5%	12.5%	25.0%	0.0%	25.0%	0.0%	100.0%
	25.0%	16.7%	11.1%	40.0%	0.0%	22.2%	0.0%	17.8%
Cross-reference	–	2	2	–	–	1	1	6
	0.0%	33.3%	33.3%	0.0%	0.0%	16.7%	16.7%	100.0%
	0.0%	33.3%	22.2%	0.0%	0.0%	11.1%	20.0%	13.3%
Catalogs	1	1	1	1	–	1	–	5
	20.0%	20.0%	20.0%	20.0%	0.0%	20.0%	0.0%	100.0%
	12.5%	16.7%	11.1%	20.0%	0.0%	11.1%	0.0%	11.1%
Accept broken	–	1	–	–	1	1	–	3
	0.0%	33.3%	0.0%	0.0%	33.3%	33.3%	0.0%	100.0%
	0.0%	16.7%	0.0%	0.0%	33.3%	11.1%	0.0%	6.7%
Auto.complete	1	–	–	–	1	–	–	2
	50.0%	0.0%	0.0%	0.0%	50.0%	0.0%	0.0%	100.0%
	12.5%	0.0%	0.0%	0.0%	33.3%	0.0%	0.0%	4.4%
Nav hotkeys	1	–	–	–	–	–	–	1
	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%
	12.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.2%
Total Sample	8	6	9	5	3	9	5	45
	17.8%	13.3%	20.0%	11.1%	6.7%	20.0%	11.1%	100.0%

Multiple responses for all abilities missed by multiple responses to selection criteria, ranked by overall frequency of mention, with row and column percents. Shaded cells show frequencies of 10 and over.

3.1.4 Conclusions

This survey clearly pointed to the interface as the locus of decisions on recommended editing software, especially with projects involving the introduction of structured editing for the first time.

- The principal findings showed that experts tended to recommend software whose interface was familiar to the users, rather than that which was best suited to the tasks, as this approach reduced the training requirements. 67% of participants tended to recommend editing software which the users were already familiar with, where possible, even if this conflicted with software which might be more appropriate for the tasks. The reasons given were that this approach minimised the need for [re-]training. A different set of editors was preferred by experts for their own use.
- For their own use in working on XML projects, participants preferred *XMetaL* (47%), *Emacs* (27%), and *Epic* (13%). It should be noted that there are additional products available since the date of this survey, which are covered in the Software analysis (section 3.2 on page 133).
- None of the experts interviewed identified any specific major feature of any editor which made it preferable to its competitors (there were a few minor preferences related to platform preferences, specific vendors, and ability to handle specific features such as Notations limited to particular vertical markets, but nothing relating to the core activities of editing XML or LaTeX).
- However, interviewees identified a number of major deficiencies, including interface clutter, crashing or hanging on large files, lack of support for external entities and for catalogs, general instability, and poor typographic control.
- The editors used or recommended were generally regarded as 'the best of a bad bunch'. This dissatisfied view must be taken in the context of the status of the participants as extensively experienced in markup systems and thus liable to be critical far beyond the scope of the average user.

In general, this survey showed a depressing lack of enthusiasm for editing software. The group is by its nature well-informed as well as highly critical, and in some cases arguably more expert in handling structured text than the manufacturers of the software: a common criticism was that some vendors appeared to be unaware of the requirements of a structured text editor.

This may be partly due to the tendency explained in section 3.3.5 on page 174: as XML matures and becomes more widespread, its business appeal becomes less interesting than the 'latest hot property' that it was in 1995. Corporate goals appear to move away from the technology itself towards more generic support for business processes.

3.2 Software analysis: the reality — what users have to work with

There is an unknown number of XML editors on the market. In a widely-used study by van den Broek (2005), no estimate of numbers is given, but the 20 initially selected for our analysis included all those known to be in common use at the time (including some which went out of existence in the following few years). Editors for other structured document systems such as LaTeX were not included in van den Broek's study.

For the purposes of the current analysis, however, a slightly narrower selection was used (12 editors, listed in Table 3.9 on page 136). This was limited to those which were suitable for editing structured *text* documents (as opposed to those more suited to 'data' applications; see section 1.1.3.3 on page 22) and which were available and installable by download without special request to the vendor, except in a few circumstances where the publicly-available or demonstration version was restricted in features.

The limitation to suitability for structured text documents was of course a prerequisite for the present research, but anecdotal — even first-hand — evidence was noted during the examination of posts to newsgroups and mailing lists in the Requests Analysis (see section 3.3 on page 161) that some authors do indeed use box-format or tabular-format editors for writing what would normally be considered continuous-text structured documents (this was later borne out by a number of such editors mentioned in the User Survey in Figure 3.22 on page 192). In fact it was noted from their messages that their experiences with such editors were the cause of their query — possibly from ignorance of the existence of any other form of editor — and perhaps a good indication of the depth of the division between the 'data' and 'document' camps referred to in section 1.1.3.3 on page 22.

In terms of usability, data-type editors by their nature tend to provide no affordances for the entry or markup of mixed content. Programmers exposed only to the application of XML to rectangular data, and suddenly faced with a requirement for semantic or visual *inline* markup in documentation, can be seen to have difficulties in finding a solution within the constraints of the software at their disposal (questions on `comp.text.xml`, *passim*).

> XML is all about owning your own data — you own it as author. The

> application doesn't own it and neither does the programmer.
>
> As a result XML is not a close fit to any programming language, and 'hard-core' programmers tend not to like it. (Quin, 2013)

3.2.1 Background

As we saw in section 2.3 on page 91, software for writing or editing structured documents *as we have defined them* has been available since the 1980s. Batch editors, line editors, and visual editors were in widespread use for all kinds of editing before this time, but facilities for handling structure with markup in the many systems derived from early formatters such as *Script*, *Scribe*, or *RUNOFF* were not embedded in the editors: this was the preserve of the early proprietary wordprocessors of the 1970s such as Wang. From 1980 onwards, however, work on the separation of content from format (Reid, 1980b; Goldfarb, 1990) paved the way for SGML, and later, XML.

Quint (1989) explains that '[i]t was simpler, in the early days, to apply the concept of structured documents to formatters. It was only once formatters of this type had proved their worth, and the concepts of logical structure had been better understood, that the first developments towards structured interactive systems were undertaken' (p. 54). Of the numerous systems mentioned at that point by Quint, only two, *GRiF* and *Interleaf*, are even recognised now by name; and both are obsolete.

In this author's own enquiry, the earliest editors designed specifically for SGML were *GRiF* (Quint & Vatton, 1986) and *Author/Editor* (Bray, Sperberg-McQueen, Harvey, Beeton, & Price, 2009, *pers. comm.*), although André, Brüggemann-Klein, Furuta, and Quint (1994) note that *Publisher* (*qv*, on page 82) 'support[s] structured documents in the WYSIWYG fashion'.

The US military and other countries' armed forces were early adopters of SGML, specifying that documentation be written to their own sets of DTDs, including the transformed output (in the US at least) with the Formatting Output Specification Instance (FOSI), notorious among developers for its complexity. A small industry grew up supplying specialist editing software for these 'military standard' applications, at what were cynically called 'military standard' prices. Adoption in the other interested areas of industry and academia was therefore severely

limited: this author recalls being quoted $5,000 per copy for an editor (single-user licence for a university Unix workstation) in 1991 — more than five years' funding for the project.

Other systems later became available at more manageable prices, but it was only with the introduction and rapid adoption of XML after 1996 that suitable software could be had at a price that even individuals could afford. Some software was of course available free of charge — an SGML mode for the *Emacs* editor was available from a very early stage (Stallman, 1981; Clark, 1992), and a sophisticated DTD parsing mode was soon developed (Staflin, 1992), which in turn led to the Schema-aware nxml-mode (Clark, 2003), all of which remain available under FLOSS licenses, and continue in heavy use.

3.2.2 Software selection

In selecting software for analysis in 2005, we initially restricted the list to those programs which explicitly handled XML or LaTeX; that is, they claimed full compatibility and conformance. During the analysis some wordprocessors and typesetters were added for comparison, but these were dropped in 2009 for reasons explained below. The findings of the 2005 analysis were reported in Flynn (2006) and are discussed in section 3.2.5 on page 139.

Owing to the fast-moving nature of software development, particularly in XML, a number of the products selected were soon superseded or fell from the market during the course of the investigation; others had been omitted because a suitable platform was not available at the time. The decision was taken in 2009 to re-work this analysis, with additional and updated products. The testing of other typesetting and wordprocessing systems was abandoned as no longer relevant in a more sophisticated market for markup products, and because it had been found unproductive to attempt comparisons between product classes which bore little relation to each other and shared few if any common features of interest.

The 2005 list had been partly informed by the work of van den Broek (2005) mentioned above, supplemented by the known availability of products advertised or displayed at the exhibitions or vendors' tables at XML and TeX conferences, and the regular announcements of software in the online forums (on page 161). The 2009 list was smaller due to the omission of the typesetter/wordprocessor categories, but included a number of editors which had only been in their infancy in 2005 (see Table 3.9 on the next page).

Table 3.9: Software analysis: products examined

	2005	2009
XML Editors	Emacs/psgml	Emacs/psgml
	epcEdit	epcEdit
	Epic (Arbortext)	Arbortext
	Exchanger	Exchanger
	FrameMaker/XML	FrameMaker/XML
	Word-11/XML	Word 2008/XML
	Excel-11/XML	Xopus
	InfoPath	XML Mind
	WordPerfect/XML	WordPerfect/XML
	XML Spy	XML Spy/Authentic
	Authentic	Serna
	XMetaL	XMetaL
		oXygen
LaTeX Editors	Emacs/LaTeX	Emacs/AucTeX
	LyX	LyX
	Scientific Word	Scientific Word
	Textures	Kile
	WinEdt	WinEdt
		TeXshop
		TeXnicCenter
Non-XML/LaTeX	QuarkXpress	
	3B2	
	Nota Bene	
	Word-11	
	AbiWord	AbiWord
	OpenOffice	OpenOffice

The two FLOSS wordprocessors, *AbiWord* and *OpenOffice* were retained in 2009, as they both possess facilities to export documents to XML (DocBook) and LaTeX format.

In addition to the differences in the sample, the only other major difference was that whereas the 2005 analysis just recorded the existence of functions, the 2009 analysis measured each editor on the functions known to be available, by recording how many keystrokes or mouseclicks were needed to reach the function from the rest position. The methodology for this is described in section 3.2.4 on the next page, and it was designed so that further software could be added to the dataset in future, and the statistics reprocessed, without loss of comparability.

3.2.3 Objectives

There were two principal objectives to the 2009 analysis:

1. to determine the extent of each product's ability to support structured-document editing;
2. to measure the effort required to access each function.

In 2005 only the first was set.

3.2.4 Methodology

1. Each product was installed using the manufacturer's defaults and following the conventions for the platform, exactly as a new user would install it.
2. In 2005, the markup-related functions provided by each program were determined by investigating the provisions of each menu, toolbar, or command-set which were specific to the handling of structured document text.
3. A sample XML or LaTeX document was loaded to exercise these functions. In most cases the document used was an early version of this thesis in DocBook or LaTeX format. The exceptions were those XML editors which only used W3C XML Schemas, not DTDs, or where there were no facilities for opening a document with its own custom DTD; for these, one of the vendor-supplied sample XML documents was used.
4. Two 'rest positions' were established on each occasion, one in mid-paragraph to provide a Mixed Content context, and one between two paragraphs to provide an Element Content context.
5. The availability of each function was recorded in a spreadsheet against the product. Where a previously unknown function was discovered in the process, it was added to the list. At the end of the first pass, the products previously examined were re-opened to check for the availability of any subsequently-discovered functions.

6. Each function was exercised using the relevant menu, command, or toolbar operation from one of the two rest positions: in some cases the position was not relevant, but in others a function would only be meaningful in one context or the other.

7. If the existence of a function in a product was not previously known *a priori*, or could not readily be identified by inspection of the menus and toolbars, it was recorded as 'not available'.

   [In 2009, in stricter accordance with the principle that this investigation concerns the use of the interface by non-expert users, no attempt was made to use the printed or online manuals or a web search to locate functions which were not immediately visible in the interface.]

8. In 2009 the number of keystrokes or mouseclicks required to exercise the function was recorded (whichever was the smaller, when keyboard shortcuts existed).

   [For comparability, the following were counted as one click: compound keystrokes (for example, Ctrl-K); hover menus (those which appear without the need for a mouseclick but which required a mouse movement); and text input or list navigation such as search strings, directory selection, and filename or element-name entry or completion. The highlighting of text or markup prior to exercising a function was not counted as a click.]

No attempt was made to measure non-markup-related aspects of the interface such as speed of response, use of screen space, or convenience of the menu or toolbar positioning.

In 2005, functions were *excluded* (marked as absent) which existed for another purpose, but which could have been used for markup reasons (for example, stylesheet management in *Emacs*, which as far as the editor is concerned is just the act of opening another file: it has no special relevance *to the editor* that the file is actually a LaTeX or XSLT stylesheet). In 2009, however, it was felt that this approach unreasonably penalised software which was evidently capable of performing the specified function, even though the current purpose had not originally been envisioned by its author.

In 2005, a considerable number of the software products were known from first-hand usage: identifying functions and locating them was therefore non-problematic, and it was felt that this was perhaps skewing the results towards

known products. In 2009 it was decided to apply a stricter rule (referred to as the 'rule of obviousness'), under which the identification of functions (and therefore their measurement) was limited to those which were immediately obvious to inspection (item 7 in the procedure on the facing page) or could be found trivially among the menus, toolbars, or command-completion.

We consider the effects of these changes on on page 149.

3.2.5 2005 results

Some tentative conclusions were drawn from the first analysis:

1. the terms used to label the functions varied widely between products: this could be a source of considerable potential confusion to the user;
2. while most of the editors examined possessed almost all the expected functions for direct handling of the markup, they differed in where these functions were to be found in the toolbars or how deep in the menus they were placed;
3. none of the editors was really suited to writing documents except by already-skilled XML experts.

3.2.5.1 Terminology used for functions

As the terms used by different products to describe their functions varied considerably, they have been normalised in Table 3.10 on the next page to a common set of terms conventionally used in the field of markup theory (see discussion in section 3.2.5.2 on page 144). Many are self-explanatory, and exist in many desktop applications unrelated to markup or even to the handling of documents (open, close, print, and the functions for handling tables). Others are markup-specific and are explained briefly below. The classification used here is purely pragmatic.

File handling: Functions related to opening, closing, and printing documents

- New file
- Open file
- Close file
- Save file

Table 3.10: Software analysis (2005): identification of functions by product

Function	Emacs	epcEdit	Epic	Exchanger	Frame	Word-11	Excel-11	InfoPath	WPxml	XMLSpy	Authentic	XMetaL	3B2	Emacs	LyX	SciWord	Textures	WinEdt	XPress	NotaBene	Word	AbiWord	OpenOffice
FILE HANDLING																							
new file	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
open file	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
close file	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
save file	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
print preview		x	x	x	x		x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
print to file		x	x	x	x	x	x		x	x	x		x	x	x		x	x	x			x	
validate document	x	x	x	x	x	x		x	x	x		x	x					x					
MARKUP EDITING																							
insert element	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x						
tag marked	x	x	x	x	x	x		x	x	x		x	x	x	x	x	x	x	x	x	x	x	x
rename element	x	x	x	x	x				x	x		x			x		x						x
split element	x	x	x	x			x			x		x			x						x		x
join element			x				x					x			x								x
remove markup	x	x	x	x	x		x		x	x	x	x	x		x	x	x		x	x	x	x	x
delete element	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		x	x			x
validate element	x	x	x	x				x	x	x		x											
NON-ELEMENT MARKUP																							
insert attribute	x	x	x	x	x			x	x	x		x											
edit attribute	x	x	x	x	x			x	x	x	x	x											
remove attribute	x	x	x	x	x			x	x	x		x											
check ID/IDREF		x	x						x														
insert entref	x	x	x						x	x		x		x									
insert comment	x	x	x	x					x	x		x		x	x		x			x	x		
insert MS	x	x	x	x					x	x		x											
insert PI	x	x	x						x	x		x											
raw-text edit	x	x	x	x	x		x			x	x	x	x	x	x	x	x	x	x				
TABLES																							
create table	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	
delete table	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	
insert row	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	
delete row	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	
merge row		x	x		x	x	x	x	x	x	x	x			x		x		x	x	x	x	
split row	x	x	x		x	x	x	x	x	x	x	x	x				x		x	x	x	x	
insert col	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	
delete col	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	
merge col		x	x		x	x	x	x	x	x	x	x	x				x		x	x	x	x	
split col	x	x	x		x	x	x	x	x	x	x	x	x				x		x	x	x	x	
edit table props		x	x		x	x	x		x	x	x	x	x	x		x	x		x	x	x	x	
edit row props		x	x		x	x	x		x	x	x	x	x	x		x	x		x	x	x	x	
edit col props		x	x		x	x	x		x	x	x	x	x	x		x	x		x	x	x	x	
edit cell props	x	x	x		x	x	x		x	x	x	x	x				x		x	x	x	x	
UTILITIES																							
search in markup		x	x	x			x	x		x		x								x			
spell-check element	x	x	x		x		x	x		x		x					x			x			
create stylesheet	x	x	x	x	x	x			x	x	x		x	x	x	x	x	x	x	x	x	x	
merge stylesheet		x	x	x	x				x	x	x		x						x		x		
switch stylesheet		x	x	x	x				x	x	x		x		x	x	x		x		x	x	
save stylesheet	x	x	x	x	x	x			x	x	x		x		x	x	x		x	x	x	x	
edit stylesheet	x	x	x	x	x				x	x	x		x			x	x		x	x	x	x	
WS handling		x	x							x			x						x				
WS display		x	x							x			x										
reveal/hide tags	x	x	x			x			x	x	x	x				x	x		x	x			
reveal/hide attributes	x	x	x						x	x	x	x					x						
reveal/hide entities		x	x						x	x	x	x											
tree pane	x	x	x	x	x			x	x	x	x	x		x					x	x	x		
element pane	x	x	x	x	x			x	x	x		x			x		x						
attribute pane	x	x	x	x	x			x	x	x		x											
toggle browse/print			x	x	x		x			x	x	x	x			x			x	x	x	x	
help	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
EDITOR MANAGEMENT																							
create entity		x	x						x	x		x											
edit entity	x	x	x						x	x		x											
notations	x	x	x						x	x		x											
assign processor			x								x	x	x				x		x				
edit metadata		x	x			x			x	x	x	x	x				x		x	x	x	x	
set character encoding	x	x	x						x	x			x				x		x		x	x	
register plugin			x						x			x		x									
table equivalence			x					x		x													
macros/scripts	x	x	x			x	x			x	x	x	x	x		x			x	x	x		

- Print preview
- Print file
- Validate document: the testing of a document against a DTD or Schema to check that only the permitted markup has been used, and only in the permitted positions.

Markup editing: the key functions that enable markup to be applied

- Insert element: the introduction of a new instance of an element type into the document; for example a new paragraph or list, or (within text) a highlight like emphasis or an identity classification like a city name.
- Tag marked: the application of an element type to existing text or markup, sometimes called 'surround'.
- Rename element: changing the name of an element type without changing the text or markup which it contains; for example changing a bulleted list into a numbered list, or (within text) changing emphasis to strong emphasis.
- Split element: creating two elements of the same name from one; for example splitting one paragraph into two.
- Join element: the reverse of split.
- Remove markup: taking away the enclosing start-tag and end-tag; for example from around some text so that it ceases to be separately marked, or (in element content) removing the list and item markup so that the items become plain paragraphs.
- Delete element: removing the markup *and* everything it contains.
- Validate element: perform validation (see the seventh item in the list elsewhere on this page) on an individual element (for example, a section) as proposed to the whole document.

Non-element markup: covers the application and management of metadata in attributes, Processing Instructions, comments, etc

- Insert attribute: add an attribute to an existing element to describe its function or identity.
- Edit attribute: edit the value of an existing attribute.
- Remove attribute: delete an attribute from an element.
- Check ID/IDREF: test that the ID referred to by an IDREF (cross-reference) actually exists; or (where possible) dereference the link in order to generate a textual reference such as a section number

or bibliographic citation.

- Insert Entity Reference: add a reference to an external or internal entity such as a special character or an image.
- Insert comment: add a comment which will be visible to the author or editor but which will not be part of the document text.
- Insert Marked Section: add text which will not be treated as markup (commonly used in documentation to allow the representation of actual markup without risk of its being interpreted as such).
- Insert Processing Instruction: add a special instruction to the subsequent program[s] about how to handle a part of the document.
- Raw-text edit: allow the markup to be edited as if it were actually text; this allows the writer or editor to trespass upon the special characters like < and & which are normally either invisible or managed by the software.

Tables: markup for tabular setting: most of these are identical to their counterparts in non-XML/LaTeX software such as wordprocessors

- Create table
- Delete table
- Insert row
- Delete row
- Merge row
- Split row
- Insert column
- Delete column
- Merge column
- Split column
- Edit table properties
- Edit row properties
- Edit column properties
- Edit cell properties

Utilities: functions that relate to the handling of the document, or to the (non-markup) qualities of the text

- Search in markup: the ability to limit a search to specific elements
- Spell-check element: limit spell-checking to a specific element
- Create stylesheet: start a completely new design and layout for an existing document.

- Merge stylesheet: combine two or more stylesheets, with rules to handle conflicts between specifications for the same object which exist in more than one stylesheet.
- Switch stylesheet: redraw the view of the document using a different stylesheet.
- Save stylesheet
- Edit stylesheet: edit the existing or prevailing styles.
- White-space handling: how to handle possibly extraneous white-space when opening a document.
- White-space display: whether or not to show possibly extraneous white-space that is created in an already-open document.
- Reveal/hide tags: controlling whether the markup is actually shown on-screen or not, and in what form.
- Reveal/hide attributes
- Reveal/hide entities
- Tree pane: display or hide a tree-diagram of the existing document markup structure.
- Element pane: display or hide a pane showing the relevant markup available at the current cursor location.
- Attribute pane: display or hide a pane showing the attributes of the current element.
- Toggle browse/print: similar to switching stylesheets, but using a browser to display a HTML rendering of the document, or a typographic formatting (often PDF) of the printable view.
- Help

Editor management: functions which relate to the editing software's ability to handle the way a document is processed

- Create entity: create a new character or external entity declaration.
- Edit entity: modify an existing character or external entity declaration.
- Notations: create or modify `NOTATION` declarations.
- Assign processor: add or change the external processing or formatting engine used to handle the document formatting or conversion.
- Edit metadata: create or edit metadata about the document as a whole (as distinct from individual attributes on elements in the body of the document).
- Set character encoding: specify a character encoding for the

document, such as UTF-8 or ISO-8859-1.

- Register plugin: add a new plugin to the editor.
- Table equivalence: convert tables from one model to another (for example, HTML, SAS Output (SASOUT), or Continuous Acquisition and Life-cycle Support (CALS)).[2]
- Macros/scripts: create or edit subprograms to handle special processing requirements.

We will return briefly to the problem of terminology and labelling in section 3.2.5.2, and in more detail when we consider how the interface might be presented (see Chapter 4). For the present, the foregoing list serves as a marker for the key functions — particularly for Markup Editing functions — without which a dedicated XML or (*mutatis mutandis*) LaTeX editor cannot work effectively.

3.2.5.2 Homogeneity

The most surprising result was the homogeneity of the list of markup-specific functions (those identified in 'Markup editing' in the list above (page 141)). Some variability had been expected, given the intense competition between products, but in fact most products investigated showed a remarkably similar list of functions. The exceptions were either target-based (LaTeX editors and wordprocessors/typesetters would not be expected to have facilities for XML-specific functions, for example), or were relatively trivial (absent in products less heavily used for text documents).

In Table 3.10 on page 140, if we exclude the non-XML-specific software (*3B2* to *OpenOffice*) and *Authentic* (at the time more a formatter rather than an editor *per se*), the only significant gap in Markup Editing functions is in the Join element function (only available in *EPIC*, *Excel-11*, and *XMetaL*). Other gaps are largely in the Microsoft products, which are little used for direct XML editing, and in *FrameMaker*, which although XML-conformant, is also more of a formatter than an editor:

- the Split element function is missing in *FrameMaker*, *Word-11*, *InfoPath*, and *WordPerfect/XML*;
- the Rename element function is missing in the three Microsoft products;

[2] The SASOUT and CALS tables specifications are heavily used in industry because of their adoption by other software systems (Sampson, 1996; Bingham, 2001); the Statistical Analysis System (SAS) software package gave SASOUT its name.

- the Validate element function is missing in *FrameMaker*, *Word-11*, and *Excel-11*;
- the Remove markup function is missing in *Word-11* and *InfoPath*;
- the Tag marked function is missing in *Excel-11*.

This means in effect that all the XML editors tested were essentially the same XML editor; that an author who understood markup and what to do with it could pick any one of them, open an XML document, and start writing or editing in the knowledge that almost all of the functions necessary for the task were present.

For the LaTeX software there was a little more variability in the handling of markup, but the list of applicable functions was much smaller, largely restricted to adding and removing markup. Despite the fact that the list of inbuilt functions ('primitives') in the underlying TeX language is many times greater than those implied for XML — over 300 in TeX compared with eight functions identified in 'Markup editing' in the list above (page 141). The set of functions required by a LaTeX author is necessarily much smaller, partly because the LaTeX interface is designed expressly to hide the programmatic complexity of raw TeX from the conventional user, and partly because we are explicitly not considering the mathematical interface, which provides a significantly larger number of very specialist operations which are out of scope for this research.

To some extent the homogeneity should have been deducible beforehand in the case of XML, as the majority of the functions are descended from mandated aspects of the XML specification, which in turn are inherited from the SGML standard. That is, if markup is implemented by enclosing material in a start-tag/end-tag pair, then one must expect to find a function in all XML software which will do this, both by allowing the creation of an empty element ready to be filled with text or further elements nested inside it (Insert element), and by allowing existing text to be surrounded by the tags (Tag marked). Nevertheless, it was useful to find that — for most practical purposes — the functions necessary for conducting the marking-up of a document are present in most of the products examined.

By extension, if this finding were more widely-known, it might significantly level the playing-field between products, as the only document-oriented terrain on which they could then compete would be ease of use — other essential business criteria such as cost, platform, support, or integration with other software from other vendors are not themselves directly document-oriented.

However, while the products may not differ significantly from each other in the functions they provide, there were differences between products in the placement and naming of those functions and menus in the interface. It is of course possible that the designers and marketers of the software came to certain conclusions about their perceived markets, and what the perceived users were deemed to need or want, and that the interface designs of the products reflect these perceptions.

Naming in particular is important where the set of possible functions is large, and the domain of expertise is strictly governed by a standard. Within the specialist area of markup theory, there is a well-established set of terms for these functions, which uses terminology from the related fields of computing science and information retrieval. These terms are used here without apology, but it is important to note that they will almost certainly be irrelevant, meaningless, or (perhaps worse) misleading for authors in the wider community outside these areas. We will discuss naming and labelling in more detail in Chapter 4.

3.2.5.3 Suitability

The third finding, and the one of greatest initial concern to this investigation, was that no single editor examined could be said to be suitable for the writer who is not already expert in XML or LaTeX. A significant understanding of markup theory, its terminology, and the specific markup (element type names) for the user's application, would all be needed before an author outside the XML or LaTeX field could even begin using the programs examined.

Some editorial facilities, including some of those mentioned in the Expert Survey (Figure 3.11 on page 127) such as XML Catalogs, *Word*-to-XHTML import/export, and typographic support, are entirely missing in some editors unless programmed in with scripts or macros. Unlike wordprocessors and DTP systems, which generally work straight out of the box, XML editors usually require customisation before they can be used for a specific application, unless the application happens to be one of the handful shipped precompiled with the product.

This is slightly less true of LaTeX, as the current default installations of popular platform distributions (TeXLive, ProTeXt, and MacTeX) include either a large selection of fonts and packages (plug-ins) or a transparent method for adding them from the Internet as and when needed.

For XML editing, the absent features noted *en passant* in most systems were:

- a realistic working set of up-to-date DTDs and Schemas with stylesheets for popular applications;
- real-time resolution of ID/IDREF cross-referencing, so that creating a reference prompts the user to identify the target, assigns an ID if none is given, adds the IDREF at the point of reference, and generates a suitable stylesheet-driven textual reference (for example, the required number, letter, or symbol);
- promotable and demotable block editing of elements in element content, so that a subsubsection moved to a section location becomes (in the process; and at user option) a section, and is not barred from the move by an error message or an audio beep on grounds of invalidity in the document structure;
- prompted visible cues for the compulsory elements (plural) of any newly-inserted element in element content[3]
- configurability of all menu items so that those irrelevant for a given application can be moved out of the way;
- use of deductive logic for the control of next-element insertion;
- appropriate handling of markup (perhaps in another syntax) which is present in material copied from elsewhere and pasted into the document;
- full control of the real-time declaration and use of internal and external (file) entities;
- usable writing tools (spellcheckers, thesauruses, grammar-checkers) sensitive to the presence of the markup.

We will be referring to many of these in the User Survey analysis in section 3.4 on page 176 and in the later proposals for model development in Chapter 4.

3.2.5.4 Miscellaneous findings

All the editors had fairly comprehensive tables editing controls, either for the HTML or the CALS table model. The more advanced systems and those with a

[3] A few systems can add a highlighted mnemonic (known as a **ghost**) to automatically-inserted elements as a reminder to the user that they must type their own text there. Despite having been around since the days of the *GriF* (SGML) editor, this feature does not appear to be common, although it is available in LyX (see Figure 4.9 on page 248), *XMetaL* and some DITA editors.

strong SGML document heritage (for example, *Epic*, *XMetaL*) were able to do both, and even more if programmed (the SASOUT table model, for example). *Emacs* has an add-on plaintext table editor in the tables-mode of the `table.el` module, which can produce LaTeX or HTML table markup; *WinEdt* has an optional tables plug-in which was also used.

The widest variations were in the 'Utilities' and 'Editor Management' classifications. While some of these are 'comfort features' added to smoothen the author's experience, some of them are critical to the operation of an editor for structured text in classical continuous-text documents (for example, entity management in XML), and their omission can only be seen as confirmation by the manufacturer that their product is not suitable for authorial use on those types of document. A number of products are indeed designed principally for the programmer or developer working in processing languages such as XSL Formatting Objects (XSLFO) or XSLT, and have only minimal features for authoring, but with the exception of *XML Spy* they were not included in the sample.

3.2.6 2009 results

As described in section 3.2.4 on page 137, the reprise of this investigation in 2009 changed the sample of editors (see Table 3.9 on page 136), and extended the nature of the data collected to include a simple measure of user effort.

Some additions were made to the functions shown in section 3.2.5.1 on page 139 as an extra category 'Behaviour', informed by some of the findings of the User survey (see section 3.4 on page 176):

Behaviour: Specific keys were tested for their behaviour in mixed and element content:

- Enter (Return)
- TAB
- Backspace
- Delete

Four additional tests were not related to specific keystrokes or menu items but particular ways of handling the Insert, Indent, and Cut/Paste functions (see section 6.3.1 on page 342 for details):

- Smart Insertion (SI)
- Target Markup Adoption (TMA)
- Block edit/move
- Block promote/demote

These last two are nevertheless shown in Table 3.11 on the following page under 'Markup editing', as the have been a part of the normal editorial process for decades, whereas SI and TMA (see section 1.1.7 on page 39) are of very recent origin — and as it turned out, apparently only available in three products at the time of writing.

One minor deletion was made to the list of functions: Table equivalence was dropped, as the requirements to convert from one table model to another appear to have been superseded by the abilities of the graphical table editing view to accommodate the features of both common models (HTML and CALS).

There appears to be no standard term for that quality of an interface which we are measuring by the number of clicks a user has to go through to reach a particular function. The term 'distance' is normal for the measurement itself (Hutchins, Hollan, & Norman, 1985) but we will use **reachability** for the *degree* to which an interface permits short-distance access to its functions (Tamir, Komogortsev, and Mueller (2008, p. 49) simply call it 'Effort from Keystroke/Mouseclick' but in our case 'click/keystroke distance' seems more appropriate).

The data for the 2009 measurements are shown in Table 3.11 on the following page.

Apart from the changes in the list of software examined, there were 206 discrepancies between the functions marked as present or absent in the software tested in 2005 and whether or not they were measured in 2009. The majority are due to the changes in approach noted at the end of section 3.2.4 on page 137, resulting in some functions being marked as available in 2005 but unmeasurable in 2009, and *vice versa*.

As explained earlier, the goal in 2005 had been simply to note whether it was possible to perform the function, by any means so long as it was provided as a part of the defined task of editing XML or LaTeX. This restriction was relaxed in 2009, which would account for the first type of discrepancy (47%), the 96 product/functions marked absent in 2005 appearing with a measurement in 2009 (Figure 3.13 on page 152, graph A). However, the more rigorous application of the rule of obviousness (item 7 in the procedure on page 138) appears to account

Table 3.11: Software analysis (2009): identification of functions and their click-/keystroke distance by product

	XML editors													LaTeX editors							WP	
Function	**Emacs/xml**	**epcEdit**	**Arbortext**	**Exchanger**	**Frame**	**Word XML**	**Xopus**	**WPxml**	**Spy/Auth**	**XMetaL**	**Serna**	**oXygen**	**XMLmind**	**Emacs/auc**	**LyX**	**Kile**	**SciWord**	**TeXshop**	**WinEdt**	**TeXnicCenter**	**AbiWord**	**OpenOffice**
FILE HANDLING	2	2	2	2	1	2	1	2	2	2	2	3	2	2	2	1	2	1	1	1	2	2
new file	2	2	2	2	4	6		4	3	2	4	3	3	2	2	2	2	2	1	3	2	3
open file	2	2	2	2	3	2		2	2	2	2	3	2	2	2	2	2	2	2	2	3	3
close file	1	2	1	1	1	1	1	1	1	1	1	1	2	2	2	1	1	1	1	1	2	2
save file	1	1	1	1	1	1	1	4	1	1	1	1	1	1	1	1	1	2	1	1	1	1
print preview/draft	2	2	2	4	0	2	1	2	2	3	2	3	2	2	2	1	2	1	1	1	2	2
print/export to file	2	4	2	2	2	2	1	2	3	3	2	3	2	2	2	1	2	1	1	1	4	4
validate document	2	2	1	2		3	–	3	2	2	2	1		–	–	–	–	–	–	–	–	–
MARKUP EDITING	2	2	1	2	2	2	1	2	2	2	2	2	2	2	2	2	1	3	3	–	2	2
insert element	3	2	2	2	2	1	1	3	3[1]	2	3	2	2	2	2	2	1	3	3	2	2	2
tag marked	3	2	2	2	2	2	1	2		2	3	2	2	2	2	2	1	3	3	3	2	2
rename element	3	3	2	3		2	2	3		3	3	3	3	3	1		1				2	2
split element	2	2	2	3	2		1	2	1	2	2	2	1								1	1
join element			1		3		1			2	2	2	1			2					1	1
remove markup	2	2	1		2	2	1	2	2	2	2	2	2		2	3	1				1	1
delete element	2	2	1	2	2	2	2	2	2	2	2	2	1			3	1				2	2
validate element		2	1	2	2	2		4		3		2	2	–	–	–	–	–	–	–	–	–
block edit/move	2	2	3	2				4	2		1	4	0		2							
block promote/demote												3[2]			2							
NON-ELEMENT MARKUP	5	2	3	2	4	4	–	4	1	2	3	2	2	–	–	–	–	–	–	–	–	–
insert attribute	5	3	3	2	4	4		4	1	1	3	1	2	–	–	–	–	–	–	–	–	–
edit attribute	5	3	3	2	4	4		4	1	1	3	1	2	–	–	–	–	–	–	–	–	–
remove attribute	5	3	3	2	4	4		4	1	1	3	2	2	–	–	–	–	–	–	–	–	–
dereference ID/IDREF		2	1											–	–	–	–	–	–	–	–	–
insert entref	Y	4	2					4	3	2	3	2	3	–	–	–	–	–	–	–	–	–
insert comment	Y	2		3		2		4		2	2	2	3	1	3	2		1		2	–	–
insert MS	Y	2		3				4	2	2		2		–	–	–	–	–	–	–	–	–
insert PI	Y	2						4	2	2	2	2	3	–	–	–	–	–	–	–	–	–
raw-text edit	Y	2		2					1	2	2	1		0	2	0		0	0	0	–	–
TABLES	2	2	1	–	2	3	3	3	1	2	3	3	3	3	1	2	2	3	1	–	1	1
create table	6	3	3		2	2	2	2	3	3	2	2	2	4	2	3	2	3	2	3	2	2
delete table	Y	2	2		2	2	2	2	2	2	2	2	1						2		2	2
insert row	2	1	1		4	3	3	4	1	3	2	3	3	3	1	1	3	3	1		1	1
delete row	2	3	1			3	3	4	1	2	3	3	3	3	1	1	1	3	1		1	1
merge row	2	2	2		2	1	3	2	1	2	3	3	3	3		2					2	2
split row	2	2	1		2	1	3	3	1	2	3	3	3	3	1	2					1	3
insert col	2	2	1		4	3	3	4	1	3	3	3	3	3	1	1	3		1		1	1
delete col	2	2	1			3	3	4	1	2	3	3	3	3	1	2	3		1		1	1
merge col	2	2	2		2	1	3	2	1	2	3	3	3	3	2	2					2	2
split col	2	2	1		2	1	3	3	1	2	3	3	3	3	1	1					2	3
edit table props		3	3		2	3	3	3	1	2	3	2			2	2	2					
edit row props		3	3		2	4	3	3	2	3	3	2			2	2	2					
edit col props		3	3		2	4	3	3	2	3	3	2			2	2	2					
edit cell props		3	3		2	4	3	3	2	3	3	2			2	2	2					
UTILITIES	2	2	2	1	2	2	–	2	1	2	2	1	2	–	2	1	2	–	1	1	–	–
search in markup		2	2	5	4					4	2	2		–	–	–	–	–	–	–	–	–
spell-check element	8		2		4	2	2	2	2	2		2	2	3		2			4	2	3	2
create stylesheet		0	2					2						0		–						
merge stylesheet		4	3											0		–						
switch stylesheet		4	3											0		2	2					
save stylesheet		3	2					2						0		–						
edit stylesheet		1	2					2						0		–						
WS handling	2	2								3				–	–	–	–	–	–	–	–	–
WS display	2	3															2				–	–
reveal/hide tags	2	2	2					2	1	2	2	1		3	2						–	–
reveal/hide attributes	2	2	2		2	2			1	2	1	1				–	–				–	–
reveal/hide entities		2	2						1	2	2	1				–	–				–	–
tree pane	2	0	0	1	2	2			0	2	0	1	2	2	2	1			1			
element pane		0	1	1	2	2		1	0	2		1	2			1				2	–	–
attribute pane	2	1	1	1	2	2		1	0	2		1	2			–					–	–
toggle browse/print			0					2	1	2	3	1		3	2	2	2			1	–	–
help	1	1	1	1	1	1	1	1	1	2	1	1	1	2	1	1	1	2	1	1	1	
EDITOR MANAGEMENT	–	2	2	–	–	–	–	4	–	4	3	–	–	–	–	–	3	–	3	–	–	–
create entity/macro	Y	9	2					4		4	3					–	–		3			
edit entity/macro	Y	4	2					4		4	3					–	–		3			
edit notations	Y	2												–	–	–	–	–	–	–		
modify processor										4				3	2	2	3	1	3	2		
edit metadata	Y	2														–	3				2	2
set character encoding	5													6		–						
register plugin/script		3									2		2			–						
macros/scripts	Y	3	0										3						3			
BEHAVIOUR																						
Enter	B	B	S			B	S	P	S	S	S	S		B	S	B	B	B	B	B	S	S
TAB/Space	I	T	P			T	E	T	P			P			L	T	T	T	T	T	T	T
BS/Del at element boundary	D	W	J		X	M	J	U		J		J		X	X	X	X	X	X	X	X	X
BS/Del in element content	X	X	X		X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X
Smart Insertion			2						2													
Target Markup Adoption							0															

grey group modal value;
blank function not provided;
– not applicable in this context;
0 function automatic;

B linebreak;
D delete markup;
E demote;
I indent;
J join;
L list (!);
M mark;
P jump;
S split;
T tab;
U unmark;
X delete text;
W warning;
Y data must be typed;
[1] in mixed content; only 1 click needed in element content;
[2] DITA only.

for the second type (53%), the 110 product/functions which were marked present in 2005 not being measured in 2009 (Figure 3.13 on the following page, graph B).

The majority of the first type (functions not recorded in 2005 but measured in 2009) were due to updates of the software. The core editing functions showed only 10 discrepancies in structured systems, all of which were additional to the editors since 2005. The 28 differences in the Tables category for LaTeX and XML editors reflected the addition of a grid-view (like a miniature spreadsheet) for graphical tables editing. The figures for the Utilities category (27) reflected the addition of stylesheet handling, and the figures for the Editor Management category (13) reflected improved handling of macros and processors.

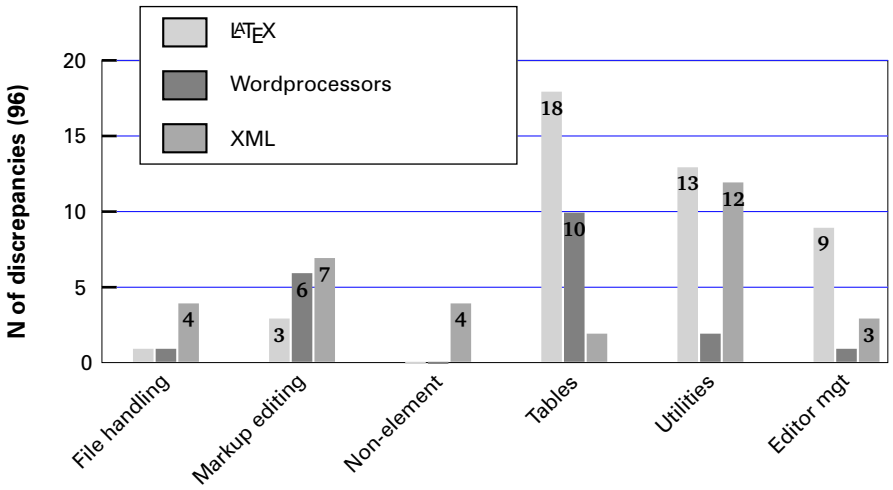
The second type of discrepancy (functions recorded as present in 2005 but not measured in 2009) showed a similar picture in the core editing functions (10), all the result of their being less than obvious. There is a somewhat different picture in the Utilities and Editor Management categories, but with the same cause: in the Utilities functions, out of 49 differences, 33 involved the obscurity of stylesheet handling functions (67%), and in the Editor Management functions (26) the majority of changes (19) were in the measurement of metadata, scripts, plugins, and encoding functions (73%): for example, a stricter definition of metadata handling was applied to mean 'metadata as provided for in the DTD/Schema' rather than system metadata stored independently by the editor.

The data in Table 3.10 on page 140 has *not* been updated to reflect these discrepancies, as the two tables represent different aspects of the functions, and the further work below is based on the 2009 data.

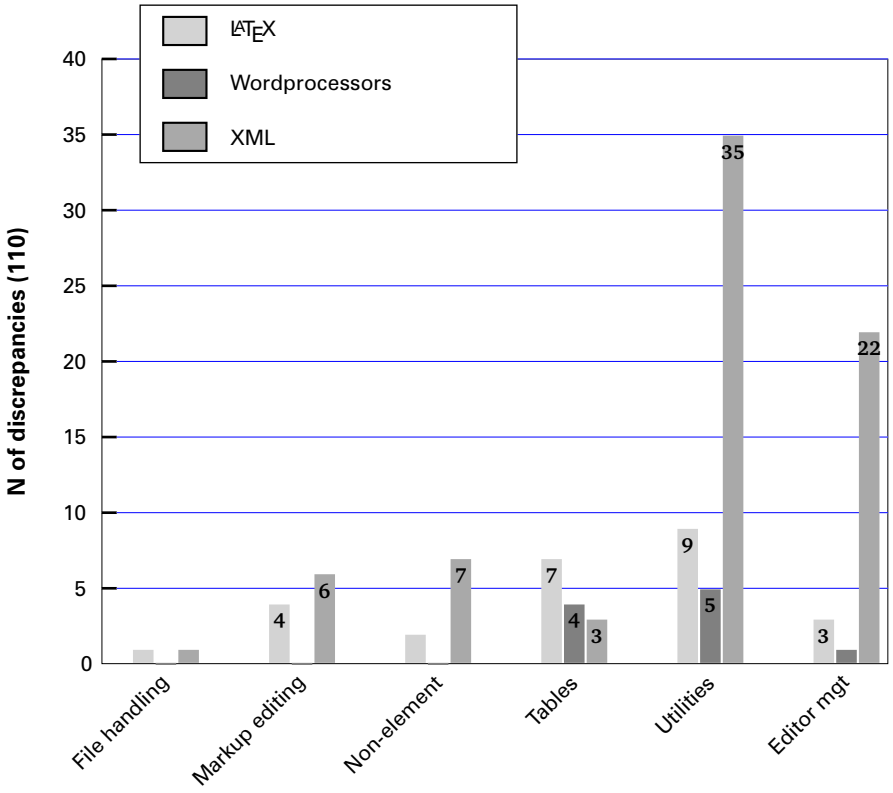
In order to provide an overview of each category, the modal click/keystroke distance score was calculated in the spreadsheet for each function represented for each product. These values are shown in the line for each category name (shaded and in capitals in Table 3.11 on the preceding page). While this is a crude statistic, and applicable only to categories and functions with measurable click/keystroke distances, it does indicates the level of effort a user can expect *as a whole* for that group of functions.

3.2.6.1 File handling

The majority of these functions took two clicks to operate, with Close file and Save file being only one click for most products.



A. Function category not recorded in 2005 but measured in 2009



B. Function category recorded in 2005 but not measured in 2009

Note: In both graphs, wordprocessor values are included for comparison only.

Figure 3.13: Software analysis: Discrepancies between 2005 and 2009 measurements

The New file function showed the most variability, with *Word* requiring six clicks, and *Frame*, *WordPerfect/XML*, and *Serna* needing four. The higher scores reflect the need in those products to enter a lengthy dialogue about what type of document to create. In most other products, one click brought up a display of the available types of document, and the second click selected the chosen type.

A frequent anecdotal complaint in the earlier days of XML editors was that the choice of precompiled types of document available was too limited (see section 3.3 on page 161). Although the actual number of document types offered by each product was not counted (the large number of different price points and industry-specific 'partner' deals means any number would be misleading), it was clear during testing that the choice is now wide, with *oXygen* even offering the TEI as a result of an agreement with the TEI Consortium; this is a document type widely used in academic and research publishing, but largely ignored by the majority of vendors.

One of the new additions to the test was the in-browser editor *Xopus* (now known as *LiveContent Create*), which differs substantially from the conventional editor. In the restricted browser environment, there is no New file or Save file function, as these operations are handled remotely on the web server. In addition, there is no separate Validate document function, as the software is not capable of creating invalid documents. Markup availability can be remotely limited, but it was the only product offering TMA by default.

At the file-handling level, the degree of similarity between XML and LaTeX editors is very high, reflecting the similarity in approach taken to the choice of document types.

The Print preview or 'draft' and the Print/export functions were common to all products, with only *Exchanger*, *epcEdit*, and the two wordprocessors requiring more than three clicks.

Validation, which applies only to XML products, was missing only from *Xopus*, for reasons explained above, and from *FrameMaker* (which is obsolescent in any case).

Overall, the most reachable file functionality was in *Xopus*, but that is a special case; the only other XML product with a modal value of one click was *FrameMaker*, but its New file and Open file values were four and three respectively. Among LaTeX editors, *Kile*, *TeXshop*, *WinEdt*, and *TeXnicCenter* all had modal values of one, and all their other scores were two (except for *TeXnicCenter*

scoring three for the New file function. The only product with a modal value of three was *oXygen*.

3.2.6.2 Markup editing

These are the core editing functions of any structured-document editor. The early assumptions (based on the unvalidated 2005 results), that all such editors have broadly the same functions, remains largely true, both for XML and LaTeX systems, but the number of such key functions is smaller than originally expected.

No function took more than three clicks, with the exception of Validate element (4) and Block move (4) in *Word/XML*. *Scientific Word* required only one click for all functions due to all options being visible on the toolbar, and *Xopus* required two only for renaming and deleting. All the remaining structured products required two or three clicks, with the exception of *Arbortext* (join, remove, delete, validate), *XMLMind* (split, join, delete), *XML Spy* (insert, split), *Word/XML* (insert), and LyX (rename). The wordprocessors *AbiWord* and *OpenOffice* need one click only for split, join, and remove.

Functions not measured were Tag marked in *XML Spy*, Rename in *FrameMaker*, *XML Spy*, and in all the LaTeX editors except *Emacs*, LyX, and *Scientific Word*. The Split function was not measured in *Word/XML* and all the LaTeX editors (but it should be noted that the commonest element, the paragraph, can be split in LaTeX syntax with the Enter/Return key alone). The Join function was not measured in six out of the 13 XML editors, and in all LaTeX editors except *Kile*. Remove was measured in all the XML editors except *Exchanger*, but only in LyX, *Kile*, and *Scientific Word* among the LaTeX editors. The Delete function was measured in all XML editors, but only in the *Kile* and *Scientific Word* LaTeX products. Validate element, however, was not measured in *Emacs*, *XML Spy*, and *Serna*: its absence in *Xopus* has been noted above for other reasons, and the function is not applicable to LaTeX editors.

Two additional functions were included in this category (see the list beginning on page 148). Block move was measured in all XML editors except *FrameMaker*, *Word/XML*, *Xopus*, and *XMetaL*, but only in LyX in the LaTeX editors. Block promote/demote, on the other hand, was measured only in *oXygen* and LyX.

Arbortext, *Xopus*, and *Scientific Word* were the only products with a modal value of one: all the others scored two except for *WinEdt*, which scored three.

3.2.6.3 Non-element markup

These functions refer almost entirely to XML editing, with the exception of the Insert comment and Raw-text edit, which are available in most LaTeX products as well.

The functions for attribute handling (insert, edit, remove) were measured almost identically for all XML editors. *Emacs* consistently took more clicks (5) due to its opening a separate buffer on each occasion, but *FrameMaker*, *Word/XML*, and *WordPerfect/XML* all required four clicks. *XML Spy* and *XMetaL* required only one click, as did *oXygen*, except for Remove.

Dereferencing ID/IDREF links, which is a key component of all cross-referencing mechanisms in structured documents, was only measured for *epcEdit* and *Arbortext*: it may well be available in others at the expense of some additional scripting, but its absence is puzzling given its trivial availability in LaTeX using the varioref package (unmeasured here, as its function is identical in all LaTeX systems and requires no additional clicks).

The position with the insertion of Entity References, Comments, Marked Sections, and Processing Instructions is very mixed. *WordPerfect/XML* required four clicks for all of them; as did *epcEdit* for Entity References; the remainder took two or three. In *Emacs* they are just typed as text, but they were measured for most XML systems except *FrameMaker* and *Xopus* (entirely unmeasured), and for *Word/XML* (comments only), *Arbortext* (entity references only), and *Exchanger* (comments and marked sections only); they were measured for the remainder except Insert comment (*XML Spy*), Insert marked section (*Serna* and *XMLMind*). Comment insertion was measured for all LaTeX editors except *Scientific Word*.

Raw-text editing, the only mode available in *Emacs*, was measured only for six of the 13 XML editors, but for all LaTeX editors except *Scientific Word*.

Modal values were more widely distributed in this set of measurements: *Emacs*/XML scored five, and there were scores of four for *FrameMaker* and *Word/XML*, and three for *Arbortext* and *Serna*. *XML Spy* was the only product with a mode of one. The measurements in this section were largely inapplicable to LaTeX systems.

3.2.6.4 Tables

Table-editing functions were by far the most consistent, and (as noted in section 3.2.6 on page 149) they almost all now use a virtually identical mini-spreadsheet editing model, whose functions work the same regardless of the host editor.

As a result there are few exceptional data points in this section. *Exchanger* has no support for tables editing beyond that provided by normal element editing, and thus remained unmeasured. Creating a table in *Emacs* (with the tables-mode referred to in section 3.2.5.4 on page 147) required six clicks; Insert row and Insert column both required four in *FrameMaker*; both row and column insertion and deletion required four clicks in *WordPerfect/XML*; and editing row, cell, and column properties required four clicks in *Word/XML*; otherwise almost all other functions required the same two or three clicks.

The notable exception was *XML Spy*, which only requires one click for all tables editing functions except creation and deletion, and properties editing, and therefore had a modal score of one. *Arbortext*, *LyX*, *WinEdt*, *AbiWord*, and *OpenOffice* also has a mode of one, but the first two and the last two had many more twos in their scoring than did *XML Spy*, and *WinEdt* was rated on much reduced functionality.

3.2.6.5 Utilities

Editing utilities vary widely between products. Most editors have a dictionary, but few structured editors come with writers' tools like a thesaurus, topical dictionary or ontological or taxonomical support, outliner, grammar checker, bibliographic support, or realtime textbase, although the more sophisticated systems are able to offer plugin support.

Even Markup search is poorly supported, being unmeasurable in *Emacs*, *Word/XML*, *Xopus*, *WordPerfect/XML*, *XML Spy*, and *XMLMind* (the function is not implemented at all in LaTeX editors). In *Exchanger* it took five clicks; in *FrameMaker* and *XMetaL* it took four; in the others, two. Spell-checking the selected element fares better, being unmeasureable only in *epcEdit*, *Exchanger*, *Serna*, LyX, *Scientific Word*, and *TeXshop*. In *Emacs* it took eight clicks to achieve (only three in the LaTeX/auctex mode); in *FrameMaker* and *WinEdt* four; and the remainder two.

Stylesheet management is not always part of an editor, as for many applications

the style cannot (and must not) be changed by an author or editor. Only three systems, *epcEdit*, *Arbortext*, and *WordPerfect/XML*, provided a measurable interface; in some others an XSLT stylesheet can be created, edited, and applied, as can a LaTeX document class or stylesheet package, but the action of modification is not synchronous with the editing of the document, so this option was not measured.

The data for white-space handling was too sparse to be useful. By contrast, the Help function was the only one to be implemented for a single click in every system tested except one (*XMetaL* requires two clicks).

The reveal/hide options for attribute editing (only really applicable to XML) are available through one or two clicks for those systems which support them. The same applies to the element tree pane, element selection pane, and attribute pane, which all operate in a broadly similar manner between those systems which support them. The Toggle browse/print, which is roughly equivalent to the View menu in a wordprocessor, followed the same pattern.

Among the Utilities, the measurements were more sparse than in other sets of measurements, and modal values are correspondingly less useful. Where there was consistency, however, *XML Spy* and *oXygen* had modes of one.

3.2.6.6 Editor management

Outside the immediate confines of the document text are the settings which apply globally: the creation and management of character and file entities, macros, plugins, scripts, NOTATIONs, processors, and metadata.

These were not measurable in most systems tested, and the results were widely dissimilar for the few which provided them. As with the non-element markup examined in section 3.2.6.3 on page 155, some of these are key features in the publishing process, and their absence from so many systems has been a point of contention with publishing users for a long time.

3.2.6.7 Behaviour

The new measurements introduced in 2009 were not done by counting clicks, as they relate to the nature of behaviour rather than to how long it takes to perform them (most are a single click anyway).

The use of these keys (Enter/Return, TAB, Backspace, and Delete) is discussed in more detail in section 4.3 on page 222. The objective in including them here was to see what range of behaviour was represented in the editors examined (Figure 3.14 on the facing page).

The Enter/Return key is inoperable (ignored) while editing in *FrameMaker*, *Word/XML*, and *XMLMind*. In the remaining systems, nine of them use it to introduce a premature line-break in the text, without terminating the current (paragraph) element; nine of them use it to split the current element in order to start a new element of the same type. One (*WordPerfect/XML*) uses it to jump forward to the start of the next element in document order.

The TAB key is inoperable in five systems; it is used by 10 systems to cause a tabbing movement to a number of spaces to the right; and in three systems it effects a jump forward to the start of the next element in document order. In *Emacs* it is used to indent the line syntactically; in *Xopus* is it used to demote an enumerated element; and in LyX (bizarrely) it is used to create a new list.

The Backspace and Delete keys are used to erase text leftwards and rightwards respectively. When, having erased all available text, they run up against an element boundary, the behaviour varies. Four systems ignore the keys in this position; ten continue to delete text (in LaTeX editors, by consuming markup; in XML editors, by skipping over the markup until more character data is found); four perform a Join function with the preceding or following element if it is of the same type as the current element. One causes demotion of the current item; one causes the element to be marked (highlighted); another causes a currently marked element to be unmarked (!); and one issues a warning.

The Backspace and Delete keys operate consistently *outside* character data content (in element content, where no text is allowed *per se*, such as between paragraphs): the cursor moves to the next available point in the relevant direction where character data is once again available.

Smart Insertion (SI) is a technique to overcome the reluctance or inability of a structured editor to insert a new element when requested, when the current cursor location indicates it would not be valid at that point. The behaviour is to move to the next location where such an element would be valid, and insert it there: details are in section 6.3.2 on page 344. To date, this technique appears only to have been implemented in *Arbortext* and *XML Spy*.

Target Markup Adoption (TMA) is a technique for purging text on the clipboard of

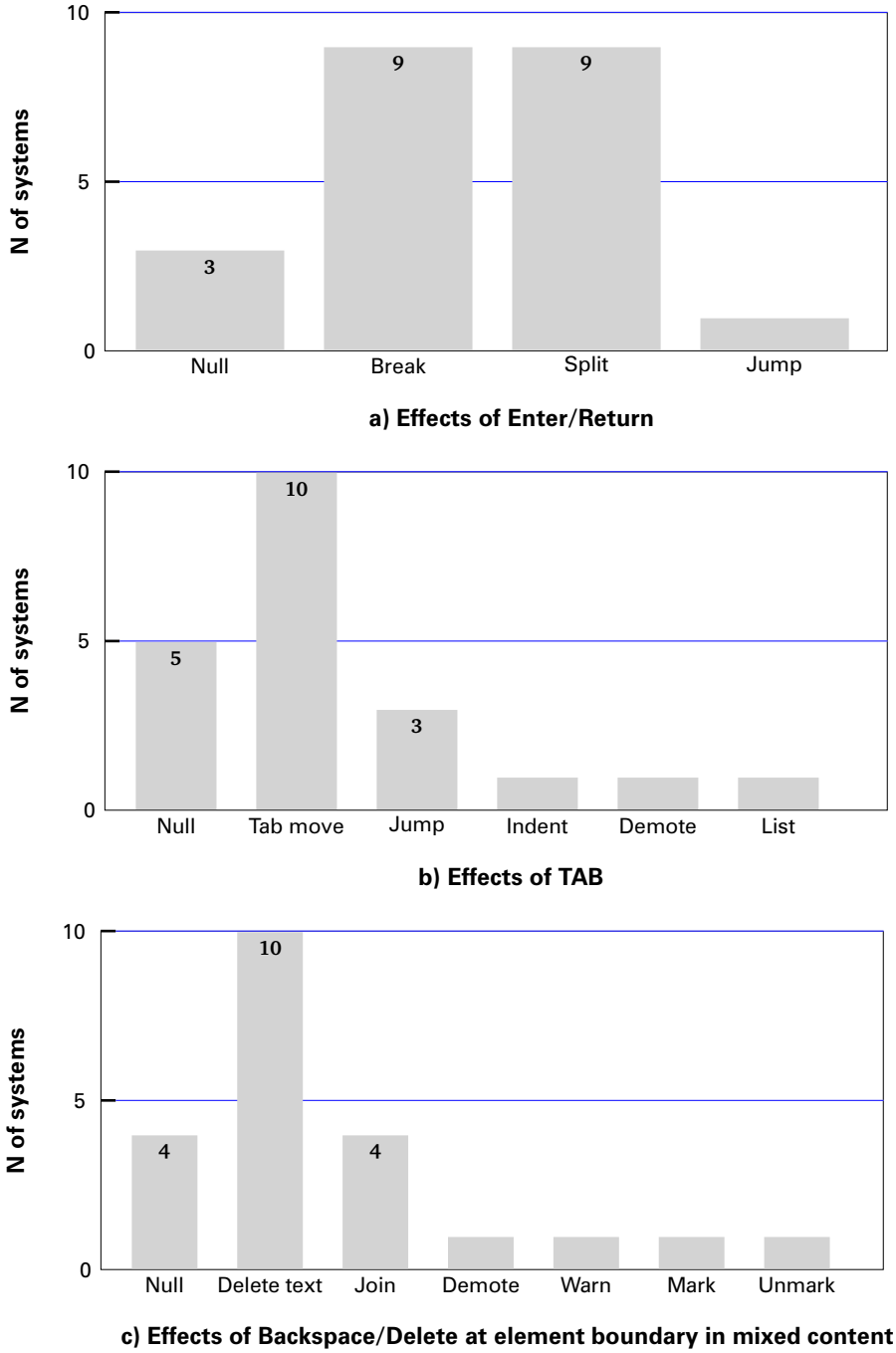


Figure 3.14: Software analysis: summary of editing character behaviour

irrelevant markup before pasting it. The method is sometimes implemented in a non-robust manner as Paste Special in non-XML/LaTeX software such as web-embeddable HTML editors, but sometimes requires a modal dialog to be completed. The method used by the sole implementor in this collection (*Xopus*) is examined in section 6.3.1 on page 342.

3.3 Requests analysis: User demand in public forums

This study tries to identify the needs of the users and potential users — what they see as lacking in the available offers.

The use of network-based discussion forums for asking and giving advice has been well-established in the computing field for many decades (Weiser, 2001). Most of them are based on the technology topic rather than on specific products (Arguello et al., 2006); where product-specific forums exist, enquiries relating to use for a particular purpose are often redirected to the appropriate technology forum.

In the two fields under consideration (XML and LaTeX), there were several principal discussion groups with worldwide coverage:[4]

```
comp.text.sgml
comp.text.xml
comp.text.tex
XML-L@listserv.heanet.ie
TEI-L@listserv.brown.edu
microsoft.xml.*
```

The first three are Usenet newsgroups, a form of distributed bulletin-board in use since before the Internet proper was established. Newsgroups are available to anyone from an Internet Service Provider, either via a newsreader application (often available within an email program such as *Thunderbird*), or from Google Groups, which maintains a web-based interface to Usenet.

XML-L and TEI-L are conventional *LISTSERV* mailing lists for the discussion of, respectively, XML itself and the TEI XML vocabulary. The Text Encoding Initiative is a set of document structures expressed in XML, principally for marking up transcriptions of literary and historical information in a form suitable for interchange between scholars, and is very widely used in research projects and publications in the Humanities (Burnard & Sperberg-McQueen, 2007). This was included principally to enable access to the views of users who were not from the natural sciences (who predominate in the three Usenet newsgroups).

[4] Some of these have since merged or moved to other forums, notably the XML discussions and much TeX material have moved to web forums. These are not so easily searched, as the pages contain a myriad of other material besides the question and answer texts.

The final set of newsgroups is devoted to discussions of Microsoft's implementations of XML in their software, and was included to provide some insight into the requirements of a population restricted by their choice of platform.

There are of course many hundreds of other forums with more specialist topics or different formats. Three particular types are evident: mailing lists, web-only forums (including wikis), and blogs. Among mailing lists, the XSL-List, for example, shares many readers with the forums selected, but deals only with the transformation of XML to other formats via the Extensible Stylesheet Language (XSL), and is therefore out of scope for this enquiry even though many of its participants would also be users of XML editing software. The purely web-based discussion groups, which cover bulletin-boards (message-boards) and wikis, are less amenable to analysis. Archives are rarely accessible, even when kept, and are either unthreaded or are not searchable with any degree of reliability. As they are individually-run, in many different formats and with no facilities for commonality of access, an overall conspectus of their content is not practicable or usable in this enquiry, despite the fact on first sight, the same types of question are being asked there as in the more traditional forums.[5] Blog articles suffer even more from inaccessibility, as their formats are generally non-compatible, and their facilities for threading, even via comments, are not accessible programmatically. Individual posts in blogs tend to be commentary rather than requests for software.

3.3.1 Objective and methodology

The objectives were to determine the quantity, frequency of occurrence, and nature of requests for relevant authoring or editing software. The method was to conduct a search in the relevant public forums for a set of keywords, and pursue the subject (or 'thread' in messaging terminology) back in time to locate the original post. These were then categorised by inspection to identify the principal nature of each request, discarding those threads where the original post was not a request (many threads wander significantly off-topic as posters comment on aspects of each others' responses).

While this method is not ideal, it has the advantage of using identifiable requests

[5] Although this is a qualitative and non-rigorous judgment, the existence of the same types of question perhaps indicates that their inclusion might only increase the absolute values for any given sample, and not affect the proportions.

from public repositories, and can therefore be repeated or updated in further work. The keyword or keyphrase method of searching is discussed in more detail in section 3.3.2.

A number of methods of categorisation were considered, including the Subject line and keyword-occurrence counting. Subject lines are notoriously unreliable, as the wording may not reflect the real content of the message, but the author's preoccupations at the time of writing or unfamiliarity with the topic. Keyword-occurrence is impractical without an authoritative thesaurus, and is unreliable when looking at posts from non-native writers of English. The use of a well-known modified Bayesian inference indexing engine (*Remembrance Agent*) was tested, but in the absence of a suitable corpus from which to proceed, this proved to be ineffective. Manual inspection was therefore used.

3.3.2 Procedure

The keywords were initially devised by inspection of recent postings to each mailing list and newsgroup asking about software for editing. The pilot keywords (stemmed manually) were:

edit
write
author
free
easy
simple
wysiwyg
best
recommend

These were tested in various combinations using the search facilities provided by *LISTSERV* and Google Groups. However, the volume of messages this method retrieved demonstrated a much wider use of the keywords than had been assumed, in many cases wholly outside the expected context, for example:

'How easy is it to... [perform some unrelated task]'
'It is simple to...'
'The author has suggested...'

Even a relatively simple Google Groups search of a single newsgroup such as `best OR recommended "editor" group:comp.text.tex` returned nearly 500 messages, and further combinations, including variant word-forms, returned a similar additional number. Many would of course have been duplicates, but the search engines available did not possess any search-set storage and provided no way to perform the union of a set of searches. Removing duplicates would have been possible by downloading the entire selection and writing a script to identify them by Message-ID (a technique used later) but this would merely have removed duplication in a set of message already noted as having too broad a scope.

When the messages relating specifically to choice of editing software were isolated by inspection, it was certainly clear that these words had a significant role, which we highlighted in the work-in-progress research note (Flynn, 2006), but the range of usage attaching to words like 'free' or 'easy' would have required extensive disambiguation *after retrieval and de-duplicating*. It was therefore decided to see if phrases, rather than individual words, might return more tractable results.

It had been noted during this stage that requests for recommendations of software tended largely to follow one or more of several patterns, for example:

> 'What [software] can I use to…?'
> 'Is there an editor for [XML]…?'
> 'Can you recommend an editor…?'

Search engines and indexing engines normally remove common words ('stopwords') to prevent them interfering with the more important (that is, topic-specific) nouns and adjectives used in most searches. In the above examples, removing the stopwords *what*, *can*, *i*, *use*, *to*, *is*, *there*, *an*, *for*, *can*, and *you* results in the two words *recommend* and *editor*, a combination which we had already discarded above as not useful. A set of phrases was therefore constructed from those messages already examined, concentrating on the actual wording of the requests, rather than the surrounding explanatory wording.

The result was a set of common types of phrase, and these were used to retrieve a second batch of messages. This naturally overlapped the first set, but resulted in a smaller overall number of messages with more specific relevance.

Information on how the commercial search engines handle quoted phrases is typically a closely-guarded secret and not publicly available, but when the phrases were split into their component words, and the stopwords removed, it became clear that this method was producing a more tractable set of data even though

they often contained the same set of keywords as the non-phrase search used for the first dataset. The words and phrases used are listed in Table 3.12.

Some adjectives (for example, 'graphical') were omitted because they already occurred in a noun phrase or compound; equally, 'author' as a verb was omitted because it was already included as a noun.

Table 3.12: Words and phrases used in retrieving messages

Nouns	**Noun phrases and compounds**	**Adjectives**	**Verbs**	**Acronyms**
editor	graphical user	experienced	input	wysiwyg
software	interface	inexperienced	edit	wysiwym
program	word-processor	easy to use	editing	gui
tool	end-user	user-friendly	write	
application	ease of use	comfortable	writing	
app	non-expert	best	authoring	
author	text-editor	free	typing	
writer			recommend	
novice			suggest	
expert				
beginner				
neophyte				
newbie				
n00b				

For the mailing lists, with publicly-available downloadable archives, it was straightforward to automate the search, identify the subject line of each message, and if necessary remove the 'RE:' prefix and locate the original posting.

For newsgroups the position was considerably less simple. Usenet is a distributed database resembling a giant bulletin-board system, constantly updated with new messages, with each server carrying its own copies of all the messages in the topics it is configured to cover. While this duplication makes local retrieval easier, limitations on local storage mean that servers are set to expire messages after a fixed time, determined by the local owner or operator. Retrieval of an earlier post is therefore often impossible after the expiry time on it has passed.

This problem was largely solved by the acquisition by Google in 2001 of the Déja News archive, which covered the period back at least to 1995; and this was subsequently expanded in some cases back into the 1980s. This extension was important: as the searches showed, some requests for software on the TEX and SGML newsgroups date back to their creation.

A script was written to use the command-line web retrieval program *wget* (Nikšić & Cowan, 2005) to query the Google Groups archive. The result of each search

was a series of web pages (each saved as a HTML file) containing one or more links to messages matching the search criteria. These pages were converted to XHTML with the *HTML Tidy* program (Raggett, Teague, Hoehrmann, Reitzel, & Vela, 2004), so that an XPath query in an XSLT script could identify the Subject and Message-ID headers for each message. A similar procedure was then followed as for mailing lists, by retrieving the original post. In some cases this was a recursive operation, as the Message-ID referred only to the immediate parent message, which might itself have been a response to another similar response, and so on back up the tree until the original was located.

Google's interface tracks successive requests, and it was found after a very short while that access from this author's computer was blocked as a security measure. Fortunately Google was very cooperative and added a special 'whitelist' permission for this author's IP address which allowed multiple requests to bypass the blocking mechanism.

3.3.3 Refining the sample

The result of these retrievals was a compendium of 5,513 messages. Of these, 3,483 messages were discarded as responses-to-responses or even lower in the hierarchy which merely quoted text from an earlier post: a sample inspection showed that their own additional content was almost entirely irrelevant. In most cases the topic had strayed, and they had been retrieved only because the text of the original post was still included as a reference. In the remainder, their inclusion in the search results was either caused by an entirely different context, or merely comments *en passant* to the topic.

For the remaining 2,030, the bulk were direct responses to the original, and could therefore safely be discarded once the original had been located. In some, the thread had become 'orphaned', so that the original was no longer identifiable, although where the original query was reproduced as a reference, the message was retained. This process reduced the number to 461. Finally, Occam's Razor was applied manually, filtering out those which did not constitute a direct request for software; that is, the request itself was subsidiary to the main topic of the message. This resulted in 403 messages for analysis, covering the period up to 2006.

An additional set of retrievals, applying identical criteria, was attempted in 2009, to try and ensure that the data also covered the years 2007 and 2008, but the

Google search applied to newsgroups had suffered from a software problem (still unresolved at the time of writing) which defeated this: attempts to repeat earlier searches returned entirely different and unrelated sets of messages with no apparent connection to the topic searched for. This has been a major source of complaints in the forums (ScooterTex., 2010), but the cause is apparently very deep-seated, and unlikely to be remedied in the near future. It would be interesting to extend this research when the problem is eventually fixed, as a general tailing-off of requests would be expected over time, as the technology becomes less novel and more mainstream, and software to handle it becomes more mature.

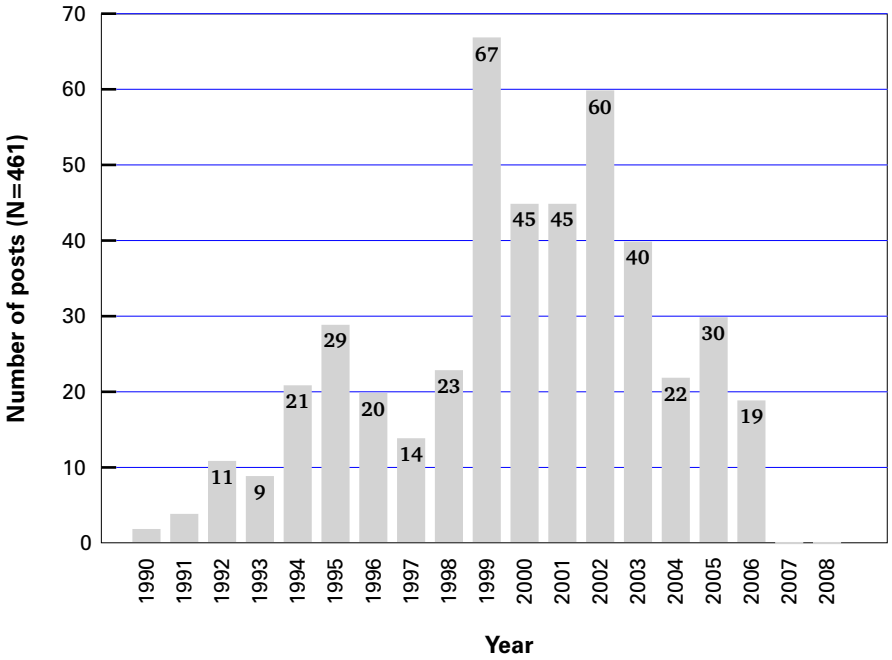


Figure 3.15: Original requests in discussion forums for editing software for structured documents, 1990–2006 (unfiltered)

3.3.4 Categorisation

Given the nature of this research, any specific mentions of usability factors in the messages were taken as overriding any other factors unless the others were given explicit priority by the poster. This meant that, for example, a message asking what software was available for editing XML, and citing ease of use, a specific platform, and free software, would be categorised as 'Ease of Use'. However, if the poster had specifically requested, for example, 'but above all, free', this would be categorised as 'Free or Open-Source' because of the explicit emphasis. Some consideration was given to methodologies described in (for example) Kirakowski

and Corbett (1990, Chapter 7), but the very restricted nature of the search — aimed at very specific topics — did not appear to lend itself to the more powerful but wider-ranging approaches.

There are possibly as many reasons for wanting software for editing text as there are people asking for it. In all cases except the simplest (for example, 'What is the best LaTeX editor?'), careful reading was required, and sometimes a second or third reading, in order to detect exactly what the user was asking for. This was particularly true in the case of posts by non-native users of English, where the mode of expression was sometimes obscure.[6] The guiding rule was to assign what appeared to be the *primary nature of the request*, even though there were in some cases other criteria of equal or unspecified importance to the user. While making a qualitative judgment of this nature is not ideal, it appeared to be better than making no judgment at all, or accepting some kind of automated classification or ranking which might or might not have been correct. Indeed, a ranking method would have introduced a level of complexity which would have placed the analysis outside the scope of this research, and an attempt to distinguish the overlapping topics proved to be intractable.

In those cases where a set of criteria was given without assigned priorities, the criteria were checked against the developing set of categories, and if a direct match was found, the message was categorised there, rather than creating a new category or classifying the message under many heads. On a second pass through the messages, it became apparent that the tone or phrasing of some messages merited their reclassification into a different category, as the overall list had now stabilised.

The final classification is given in Table 3.13 on the next page. The specifics of the categories are discussed in section 3.3.5.

3.3.5 Analysis

The general requests (that is, those without qualifications) were the single largest category, which is unsurprising given the relatively new nature of XML and markup to many people, and the fact that those who seek help from the public forums are typically the newcomers to any field.

[6] Discussion forums do of course exist extensively in other languages: there are for example `de.comp.text.tex` and `de.comp.text.xml` newsgroups for German speakers, but examination of these would have required a degree of fluency in multiple languages which was not available.

Table 3.13: Classification of messages requesting editing software for structured documents (1990–2006)

Apparent primary nature of request	**N**
General requests for editing software (no specifics provided)	65
'A WYSIWYG editor'	50
Details of a specific editor (by name)	47
Single-platform editors	39
Specific application or business requirements	30
Free, Libre, or Open-Source software (FLOSS)	28
Embeddable editors (Java, C++, VB, etc)	27
Specific DTD/Schema handling	21
Structure-view editors	20
Mathematical markup	17
'Easy-to-use' editors	16
Formatted-view editors	11
Vendor or developer offerings	9
Unicode and language support	7
Multi-platform editors	6
Strict DTD/Schema-observant editing	5
Total messages	403

It was clear, however, that the demand for WYSIWYG behaviour was the main specific request. To what extent the closely-related terms 'ease of use' and 'Graphical User Interface' are conflated with WYSIWYG in the enquirer's mind is not clear, and cannot be determined from an inspection of the messages. In the User Survey (section 3.4 on page 176), an examination of comments showed that GUI and WYSIWYG are used synonymously, although as we saw in section 1.1.7 on page 39, strictly speaking they refer to two separate traits of computer interfaces. While 'ease of use' is clearly related to these, it was separately specified in sufficient posts for it to be used as a distinct category.[7]

General requests for editing software: (with no specifics provided). These requests generally took one of two forms: a single direct question such as 'What software do you recommend for editing XML?' (39: 60%); or a more verbose form with surrounding explanations about their project or requirements (26: 40%). Nine requests in each case (25% of the direct questions; 35% of the indirect) were phrased in such a way that it was clear the poster was unaware of the existence of XML or LaTeX editors (references to 'some tool' or 'a program' to let them manage their text).

[7] A few users clearly understood that 'ease of use' did *not* necessarily equate to WYSIWYG, but it was not evident if this related to their knowledge of interfaces, or to poor past experiences.

This was the most frequently-occurring specific request. Requests for 'a GUI editor' were included in this category: as we have explained in section 1.1.7 on page 39, the two terms are frequently, if erroneously, conflated. In almost all cases, there was a demonstrable reluctance on the part of the poster to see or even be aware of (let alone learn) the inherent syntax (pointy brackets, backslashes, curly braces, etc). In a few cases the tone was even bordering on the hysterically affronted that no such editor appeared to exist, making it clear that the poster could not understand *why* there was no such editor, giving force to the view that all other considerations should be subordinated to the creation of a synchronous typographical interface.

Details of a specific editor: These were requests for more information about a named product. In these cases the user had clearly encountered the software or had been recommended it, but wanted to know if it could handle a particular task, or if it exhibited a particular feature. These were therefore not requests for suggestions for software, but they were an indication of some of the software already in use (or imminently so). This category does not include the numerous subsidiary references to a requirement that an editor be '*Word*-like'. A list of the software mentioned is in Table 3.14 on the facing page.

Single-platform editors: These were requested for Microsoft Windows (13: 33%), Unix and GNU/Linux (19: 49%), Apple Mac (5: 13%) and others (2: 5%)[8]. The higher value for Linux than Windows, despite Windows' much larger installed base, may relate to the fact that a very large amount of development done on Unix and GNU/Linux systems includes most FLOSS development, in which the pace of development is faster in the early stages, so that new systems tend to be released faster than for Windows, and thus garner more requests (Feller & Fitzgerald, 2001).

Specific application or business requirements: These requests covered a wide spectrum with no discernable common thread except that the users wanted to be able to create or edit structured documents for their own specialised application, for example in a 'vertical market' for an industry

[8]The now-defunct Digital Equipment Corporation's (DEC's) Virtual Address eXtension/Virtual Memory System (VAX-VMS) family of minicomputers and International Business Machines' (IBM) AS/400 series. The GNU's Not Linux (GNU) prefix distinguishes Linux from the original UNIX[es], which include Berkeley Systems Division (BSD).

Table 3.14: Number of mentions of specific editors in messages requesting advice

Product	Mentions	Usage
Adept/EPIC/Arbortext	9	✓
FrameMaker	7	✓
Scientific Word	6	✓
WordPerfect/XML	5	✓
Word/XML	5	✓
XML Spy	4	✓
SGML Author for Word	4	
Emacs	3	✓
XMetaL	3	✓
Author/Editor	2	*
Elsevier (*Pandora*)	1	
Emilé	1	
FrontPage	1	
Interleaf	1	
K42	1	
Leo	1	
Morphon	1	
XMLMind	1	✓
Unknown	1	

* The *XMetaL* editor is a descendant of *Author/Editor*. Other products marked in the Usage column refer to their inclusion in the Software Analysis (section 3.2 on page 133 — refer to Table 3.9 on page 136).

sector; or for a particular task such as creating complex tables, or using Regular Expressions in searches.

Free or Open-Source software (FLOSS): This was most often requested (18 times; 64%) using the term 'free' (assumed to mean 'free of cost' rather than 'free of restrictions'). By contract, Open Source, Shareware, or Public Domain software were requested 10 times using those phrases (36%). Comments *passim* about the high cost of commercial XML software possibly contributed to this demand.

Embeddable editors: These are libraries or toolkits of software components with which software developers can build their own editor to run inside another program, for example a web browser. The language can vary: Java was by far the most popular (15: 55%), followed by browser-based editors (8: 30%), with the remainder minor or unspecified (4: 15%).

Specific DTD/Schema handling: This is not a feature generally provided in conventional XML editors, as the document type is presumed to be fixed by the application, managed at a corporate or project level of control, and not

available to the author or editor for private manipulation. Editors for software developers, of course, include extensive DTD and Schema design and manipulation tools, but these editors are for application designers and cannot conveniently be used for normal text document writing by non-experts.

Structure-view editors: These cover a class of editors which display the document structure in tree form or as blocks in a hierarchy as the primary view of the document. Although the tree model is widely used in software development, the Experts' Survey felt it was largely shunned by authors even when they are aware of it, because it represented an unfamiliar model of the document. Human editors, however (and a few writers), may make use of the affordances these editors provide for block-move operations using the tree display (see section 3.2.6.2 on page 154), as it avoids the difficulty of mouse and cursor control in scrolling to mark a large block of text which spans significantly more than one element in element content.

Mathematical markup: Mathematics for LaTeX and XML was requested both separately here and as a further requirement in other categories. While graphically-driven mathematical markup is out of scope for this research (see section 6.3.14 on page 359), it is demonstrably a requirement that is yet unsatisfied: the market leader (*Arbortext*) produces XML with MathML, but is unaffordable by many; whereas the FLOSS editor LyX is cost-free but produces LaTeX files.

'Easy-to-use' editors: These are a common subsidiary requirement in other categories as well. This topic is discussed in greater detail in section 3.3.5 on page 169, as the user perception of 'ease of use' has a significant overlap with other terms related to usability.

Formatted-view editors: These are editors which provide a level of synchronous typographic editing that falls deliberately short of what is optimistically described as 'full' WYSIWYG. The requirement is for sufficient typographic rendering to enable the author or editor to interact with the document structure, but without the need for the display to represent the fine detail of the fully-typeset page.

Vendor or developer offerings: These include announcements of personal projects to develop XML or LaTeX editors (usually as FLOSS), and requests for advice on the facilities and features to include. It explicitly *excludes*

advertisements and marketing announcements of commercial products.

Unicode and language support: This has moved from a specialist requirement to a standard facility: Unicode compliance is mandated by the XML Specification, and LaTeX supports it as best it can with the ucs and inputenc packages (XeTeX and ConTeXt extend this facility and make it a default). Nevertheless there are clearly users whose first encounter with lower-grade or non-Unicode-conformant software led them to specify this as a prime requirement.

Multi-platform editors: These were perhaps unsurprisingly a low priority, given the prevalence of Microsoft Windows in most areas, despite the common perception that software discussions are the province of the programmer and designer, who might be expected to take the wider view.

Strict DTD/Schema-observant editing: This was specifically requested by some users in order to ensure absolute conformance, and users were at pains to explain that this meant draconian ruthlessness: no user interference allowed, full population of nested elements where required, and no options to choose from.

If we restrict ourselves to the domain of usability, and consider only those requests which seek features belonging specifically to the interface, the following were the most-requested features:

'A WYSIWYG editor' (50)
Structure-view editors (20)
'Easy-to-use' editors (16)
Formatted-view editors (11)
Unicode and language support (7)
Strict DTD/Schema-observant editing (5).

Applying Occam's Razor, we may collapse 'WYSIWYG', 'Easy-to-use', and 'Formatted-view' to a single category of typographically-driven editors (77). Structure-view editing, in whatever form, is thus clearly still an important category but at a much lower level of demand (20). The remaining two features are properly attributes of editors, not editing modes in their own right.[9]

[9]Unicode is in any case mandated by the XML Specification, and both Unicode and language support are already extensively provided in LaTeX. All XML editors claiming DTD or Schema conformance can at this stage be assumed to provide it, but the more complex behaviour required to maintain absolute conformance *during editing* may not be provided by all editors.

The existence of such a large class of requests for the easy, typographically-formatted, WYSIWYG editors, appears to conflict with the claims from leading editors that their products *are* easy to use, *are* WYSIWYG, and *do* format things typographically. This implies either that information about these products is not well publicised, or that the products fall short in some way. The first possibility is outside the scope of this research, but the second led directly to the need to find out from users how they used their editing software.

Lapeyre and Piez (2007) summarise some requirements and tests for XML software tools:

- Tools must adapt to users' [authors'] requirements;
- Tools must prevent validation errors;
- What expectations have tools inherited from *Word*?

In response to this last question, the three most-required expectations were in fact entirely non-markup-related: WYSIWYG, spell-checking, and change-tracking (although the last — like **overlap** — poses special problems in terms of markup, which have never been resolved entirely satisfactorily). We will return to change-tracking in section 6.4.2 on page 365.

One aspect of the demand analysis needs additional clarification: the abrupt fall-off in posts to all the forums used since this study was conducted. The tendency has been noted for once-specialist software (such as an XML editor) to become mainstream over time — this is a normal effect of a maturing technology. However, a commercial side-effect appears to be that smaller companies (or the smaller arms of larger companies) specialising in this software tend to be absorbed or acquired by larger ones as an investment, rather than with the objective of improving the software itself. There are many instances of this:

- SoftQuad, the original authors of the *Author/Editor* and *XMeTaL* editors, went through many changes of ownership and were eventually acquired by Corel (WordPerfect) and then by JustSystems. Although the core team lost many of the original developers, some remain with the product today, while the original parent company lost its way making children's games.
- *WordPerfect*, which has its own XML editor, is still available, but overshadowed by the 'Office' suite with which Corel attempts to compete with Microsoft.
- The *Arbortext* company, and its editor, which has gone through many

names, was eventually bought by PTC; while it remains available, it is no longer identifiable through PTC as an XML editor.

At the same time, the use of XML has increased enormously, especially in the 'data' mode (see section 1.1.3.3 on page 22), where much of it is machine-generated, not hand-edited. In fields such as publishing, where XML forms a key part of the production process, while its use in authoring is restricted by the lack of usable editors — as we explain in this research. We can perhaps conclude that the decline in requests for editing software is not so much a function of declining demand, as an acceptance by authors that traditional XML editors are unlikely to meet their needs. The growth of HTML5-oriented editors, which we touch on in section 6.4.2 on page 365, seems likely to renew interest in the *interface* rather than the underlying technology, which is as it should be.

3.4 User survey: the reality — how do the targeted users currently cope?

To refine the topics raised in section 3.3 on page 161 and find out what software the authors and editors used and how they used it, it was necessary to gather responses directly from a population which had sufficient exposure to the practice of working with structured documents in their own field but who would not be specialists or experts in the field of markup itself. For consistency and comparability, the same target population was used as for the Requests Analysis.

The problems of working with a survey using a self-selected sample, even from a relatively restricted population, are well-known and extensively documented in the literature on survey design and sample selection; for example, Moroney (1984). Inherent bias, self-interest, sectoral interest, sample skew, low response, and the absence of definable population characteristics make it difficult even when the population size *and* the sample size are big enough for the principle of central tendency to operate reliably. In online surveys in particular, it is virtually impossible to ascertain identity with any degree of confidence when anonymity needs to be assured, and especially when facilities and services for anonymising transmissions are freely available, and where users expect and require anonymity as a precondition for participation.

Nevertheless, in a restricted population composed largely of professionals in their own fields (including those studying for a profession), the likelihood of large-scale and deliberate falsification of responses was felt to be minimal. The inclusion of an opportunity to supply an email address to receive a copy of the survey results provides a means of testing identity (of the address itself; not of the person who typed it), and the guarantee of anonymity goes some way towards ensuring that such an address will be dissociated from the data record on which it was entered.

3.4.1 Questionnaire construction

An initial set of questions was constructed, aimed at answering the questions raised by the Requests Analysis, specifically about why users were seeking the software, why they found existing solutions inadequate, and what specific features they wanted. However, a brief pilot among volunteer colleagues showed that both the first and the second draft concentrated on the attempt to elicit too

much qualitative information ('Why do you...?' rather than 'What do you...?' or 'How do you...?') and was too complex and time-consuming to complete. A comparison with the personas developed in section 1.3.3 on page 47 showed that much of this was already in the *a priori* suppositions, and that gathering further qualitative data would not advance the investigation.

It was therefore decided to approach the question by addressing those areas of the interface known from the Expert Survey and the Requests Analysis to be particularly problematic, and ask the respondents to describe their way of performing a set of actions. By gathering data about how users went about the tasks it would be possible to construct a model of the user in their work situation. When compared with the problem areas, this could indicate where improvements could be made. The resulting set of questions is discussed below. The complete questionnaire is reproduced in appendix C on p. 381.

The available responses in each case were presented as a multiple-choice list which was constructed from the Software Analysis to represent the known affordances. An 'Other' category was also provided, although it turned out rarely to be used.

1. **What is your occupation or profession?:** *Open-ended question, single line entry box for the response rather than a multi-line edit box. For the number of responses expected (and received) there was no advantage seen in pre-coding a list of occupational categories.*

2. **What kind of organisation do you work for (if any)?:**

   Big business (corporation)
   Small/medium enterprise (SME)
   Education (school, college, university)
   Research (industrial, non-profit, Non-Governmental Organisation (NGO))
   Government (federal, state, or semi-state bodies)
   Consultancy
   Student (undergraduate, postgraduate)
   Self-employed (sole proprietorship)
   Freelance (author, editor)
   Other:

   This list was chosen to distinguish between business, educational, governmental, and personal (professional) users at various levels, in order to learn if the work

environment was related to their responses.

3. What computer system(s) do you use most often?:

Microsoft Windows
Apple Macintosh OS X
Unix (including GNU/Linux, BSD, etc)
Other:

There are still mainframe XML and TEX users.

4. What document system(s) do you use most often?:

'Dot-line' (for example, *RUNOFF*/*nroff*/*troff*)
Script/GML
SGML
XML
Plain TEX / ETEX
LATEX / ConTEXt / Texinfo
WordPerfect
Microsoft Word
OpenOffice
HTML (web pages)
Wiki
Other:

The categories are in thematic order, oldest first, to identify the experience of older respondents. The last six (barring 'Other') reflect the current proprietary and open document markup formats in popular use.

5. How long have you been writing documents on a computer?:

0–2 years
3–5
6–10
11–15
16 and over

XML was first mooted 15 years ago; the Web became available 20 years ago.

6. What types of documents have you had most experience with?:

Text documents (books, reports, articles, theses, essays etc)

Structured data (e-commerce, messaging, data interchange, configuration, etc)
Web pages (HTML, content management systems, wikis)
Multimedia (graphics, audio, video, media synchronisation)
Other:

The first two are the conventional categories of XML/LaTeX documents (see section 1.1.3.3 on page 22). The second two reflect the use of online systems.

7. **What editor do you prefer to use for structured documents:** (if you have a choice)? If you have no choice, please prefix your answer with an asterisk (*)

   This is an open-ended edit box. This question in particular could not be constrained to a list of products, but it was felt to be important to allow the subject to indicate any organisational constraint on their use.

From here until item 17 in the list below (page 184) the selection choice for each question was a multiple-choice checkbox. The reasons for the specific set of options are given after each one: the objective in these questions was to push at the boundaries of each scenario in order to gather the information required about how the software is used for each task: some of the options overlap, some are control questions, and some are included because they reveal information about the nature of the markup being handled.

8. **How do you create a new document of the right type?:**

   - There is a 'New Document' menu where I choose the document type I want
     This is present in all editors tested (see section 3.2.6.1 on page 151).
   - I open an existing document of the right type, and delete the old text
     A widespread but much-deprecated practice, as in some systems it leaves the metadata of the previous document content untouched, which may be a security risk.
   - I manually edit an empty file to specify the type of document
     This is the case with plaintext interfaces such as Emacs and many LaTeX editors.
   - There is a document type creator for new types of document I haven't used before
     Larger XML systems restrict the ad-hoc creation of new document types and require a formal process for compiling the DTD/Schema.
   - The 'New Document' menu doesn't have the document types I want to use

This was included to test the anecdotal evidence that editors are shipped with an inadequate selection of precompiled document types.

- There is no choice: I use the File|New menu and just start typing
  Wordprocessors and other non-structured editors would use this as their primary method of performing the function.
- There are sample templates I can use or modify to get what I want
  This is an extension of the restrictions imposed above on new document type creation: anecdotal evidence and first-hand experience suggests that in many cases one or more of the precompiled document types is in fact a close match for the requirements.
- I check a skeleton document out of the repository
  Added to catch those respondents working in a tightly-controlled environment where access to documents is governed by a document management system.
- Other

9. How do you give the title, author, and other key information for a new document?:

- My editing software asks me for all this as I open a new document
  Some DTDs/Schemas are designed so that required metadata will be requested as a function of the relevant element types being compulsory; other editors may ask for their own metadata.
- There are spaces to fill this in after the new document opens
  This is the alternative to the above: the element types are compulsory but that in itself is not enough to cause the editor to prompt for content.
- I type it in and format it to look right
  This would be the default for a wordprocessor.
- I type it in with instructions to identify it
  This would be the default for LaTeX (and perhaps some users' perceptions of XML).
- I have to specify this before creating the new document
  Document management systems may require their own metadata ahead of time.
- There is a separate procedure to go through for each document type
  Stricter control of metadata will only allow specific documents to be created after this has been done.
- I don't: it's determined beforehand and inserted automatically
  At the strictest level, document creation is determined elsewhere, not at the author's desk.
- Other

10. How do you tell your editor to start a new section (or chapter,

subsection, subsubsection, etc)?:

- I click on 'section' (or whatever) in the 'New' menu
  This mechanism does not exist in any editor tested: it is proposed in section 4.3.3 on page 246.
- I move past the end of an existing section and click 'Insert'
  This tests if editors provide cursor access to element content. Some do not.
- I position the cursor to the right place and type the sectioning instruction
  Manual insertion in the manner of LaTeX or plaintext-edited XML.
- There is no way to do this: I have to type the heading and format it by hand with font and size menus
  The default for wordprocessors.
- I type the heading and use the template style menu to specify the type of heading
  The more sophisticated alternative in wordprocessors.
- I type the heading in a dialog box and label it as a section
  Grid-driven editors provide this facility (for example, InfoPath).
- I position the cursor inside the end of the previous section and split it to make a new one
  This use of the Split element function is commonplace in Emacs.
- Other

11. How do you apply formatting or styling?:

- I don't need to: my editing software does it for me based on the template style
  This method is the default for any pre-styled application, including wordprocessors with stylesheet facilities.
- I don't have to: the editorial production team does all that later
  Writers without a requirement to style (or prohibited from styling) would be in this position.
- I use the font menu, the size menu, and the B, I, and U buttons (or their keyboard shortcuts)
  The wordprocessor default.
- I highlight the text and use the predefined Styles menu
  The wordprocessor user with a stylesheet.
- There are toolbar buttons for identifying stuff according to meaning which do the formatting automatically
  An alternative interface in structured editors where the range of element types in

mixed content is small.

- Other

12. How do you move text blocks around when you edit a document?:

('Blocks' here means whole paragraphs, items, lists, tables, figures, sections, etc, right up to entire chapters; not words or phrases within a block.)

- I highlight the text with the cursor, then cut and paste
  This is the non-structured approach; see section 6.3.8 on page 352 for a discussion on how this might be implemented in a structured environment.
- I mark blocks using a structure window, then cut and paste
  This detects editors which implement a tree pane or similar.
- To move whole blocks I have to move their content first, then delete the empty container
  This would be used by older systems lacking the concept of a block move.
- I mark blocks in the structure window and then click 'Move' and specify the destination in a dialog box
  An alternative interface using a tree pane, sometimes also found in web-based Content Management Systems.
- I highlight the text with the cursor, then cut and paste, but I may have to reorganise the structuring instructions manually afterwards
  This is added to catch editors which do not implement SI (section 6.3.2 on page 344).
- Other

13. How do you navigate around the document when editing?:

- There is a structure window where I can click to move to a different place
  This identifies editors which use the tree pane for navigation.
- I scroll up or down until I find the part I want to edit
  This is the default action for almost any GUI or TUI application.
- I use the search to find the word or phrase I want to edit
  A more subtle approach if the writer can formulate a sufficiently unique search string (or has Regular Expression searching).
- There are navigation buttons to let me move through the structure
  Identifies editors with Jump to functions.
- I use page thumbnail images to scroll through the document
  Editors with continuous synchronous preview and a backlink to the document text.
- I use keyboard shortcuts to jump to or cycle through the structure

An alternative for the GUI Jump to functions.

- Other

14. How do you add blocks like tables, figures, lists, sidebars, etc?:

- I use the 'New' menu to insert them
  This mechanism does not exist in any editor tested: it is proposed in section 4.3.3 on page 246.
- There are toolbar buttons or keyboard shortcuts to do these things
  These are present in many 'hard-wired' editor applications and may be configurable in others.
- I click on the 'Insert' menu and pick the one I need
  The default interface; but see section 4.3.3 on page 246.
- I type instructions to identify what I am inserting
  The default for LaTeX and other plaintext interfaces.
- I drag and drop an icon from the toolbar into the document body
  An alternative method for toolbar operation.
- I type the text, then use the template style menu to identify it
  Used in wordprocessor styles interfaces.
- Other

15. How do you create or edit linking items like cross-references, footnotes, bibliographic citations, hyperlinks, acronym references, etc?:

- There are toolbar buttons or keyboard shortcuts to do these things
  These are present in many 'hard-wired' editor applications and may be configurable in others.
- I type the instructions to identify them
  The default for LaTeX and other plaintext interfaces.
- I use the menus to insert them and then add the information in a dialog box
  Implemented by most synchronous typographic editors that handle citation/reference.
- I can drag and drop references from a reference management application
  Both main wordprocessors and most dedicated LaTeX editors.
- The structure window lets me drag and drop cross-reference points
  This can be implemented in a scriptable tree pane application if the reference is stored in the document.
- I have to create and identify the target item first, then link to it by one of the above means

This would be used in any manual system (for example, using ID/IDREF).

- Other

16. How do you know how your document-in-progress will look?:

- The on-screen formatting in the editing software is good enough
  The more experienced editor (human) will understand that WYSIWYG is not 100%.
- I don't need to: it all gets formatted by the production team later
  Used in a highly managed document environment.
- I don't worry about it while I'm writing: so long as the structure is right, the formatting will follow, and can be tweaked later
  A more relaxed attitude from users who understand structure-driven stylesheets
- There is a typeset preview window showing the updated formatted output
  Used by XML editors that use browsers or wordprocessors as their preview tool; also used by some synchronous LaTeX systems.
- There is a toolbar button or keyboard shortcut to display the formatted output
  Mostly used by asynchronous LaTeX systems.
- I click on Reload in a browser window
  Mostly used by asynchronous XML systems.
- It's not necessary at the writing/editing stage: the document will be in multiple different formats
  Complex multi-targeted document management systems will use this view.
- Other

17. What best describes your general approach to creating and maintaining structured documents?:

- I use editing software that prescribes and enforces structure
  To detect prescriptive application users.
- I use editing software that encourages and supports structure
  To detect descriptive application users (original authors).
- I use editing software that lets me describe and identify the appropriate structure myself
  To detect descriptive application users (encoders).
- I use structuring instructions only where necessary
  Minimalists.
- I don't use structured documents
  Control question

- Other

18. **When advising others (for example, colleagues, clients, students/trainees, friends) about software for editing structured documents, what specific feature(s) do you suggest they look for, or guard against (if any)?:** *Open-ended*
19. **What is the single most useful feature of the editing software that you use the most? (and what editor is that?):** *Open-ended*
20. **What is the single worst feature of any editing software for structured documents that you have used?:** *Open-ended*
21. **When using an editing system for structured documents, have you ever failed to find out how to do something that the product was actually capable of doing? Please describe the circumstances briefly.:** *Open-ended*
22. **What features would you like to see in an editing system for structured documents that aren't in any of them at the moment? (ie your WishList):** *Open-ended*

The last six questions are included to test against those asked in the Expert Survey (section 3.1 on page 107).

Two final questions asked for an email address if the respondent was interested in an anonymised copy of the results, and for any feedback on the questionnaire process itself.

3.4.2 Survey administration and processing

Participation was solicited in messages to the same forums as for the Requests Analysis (section 3.3 on page 161), but it was also publicised in the XML Frequently-Asked Questions (FAQ) site, the TEXhax newsletter, and the LATEX (Google) mailing list.

3.4.2.1 Sample size and participation rate

The survey was explicitly *not* announced to groups who would have had specialist expertise in working with markup in their occupation, such as the Society of Technical Authors; while the individuals involved have much to contribute (some of the Expert survey respondents were members), their expertise places them well

outside the target group for this research — it would have overshadowed that of users who are not technical writing experts.

Several attempts were made to widen the scope and seek the interest of publishers and of writers' organisations, but all were unsuccessful. The survey does not therefore try to represent the interests of the generality of writers but concentrates on identifying how users of structured editing software who are not markup experts use their interfaces. Further work would be required to extend the reach of this enquiry into other fields.

There were 62 valid responses to the questionnaire. An estimate of the population (those who might have seen the announcements of the survey) is difficult to make: the membership of the mailing lists was approximately 700, but the readership of the Usenet newsgroups is not knowable, and there is considerable overlap. The responses therefore represent a small interested sample: we estimate below that it is a 2% sample of those exposed to the announcement. The scope of this announcement is discussed further in section 3.4.3 on page 189.

There will have been many readers exposed to the announcement whose interests in XML were in its use for data representation, not structured text documents (see section 1.1.3.3 on page 22), and who were thus unconcerned with the topic, and therefore in any case not part of the target population. A coarse estimate of the sample percentage, based on this aspect of those two principal modes of usage, can be obtained by comparing the number of messages retrieved in the Requests Analysis in section 3.3.3 on page 166 with the estimated total number of messages in the selected Usenet newsgroups and mailing lists (see Table 3.15).

Table 3.15: User Survey: Estimate of population size exposed to the announcement of this survey, based on the extraction rate of messages in the Requests Analysis

Source	Messages	Dates
`comp.text.sgml`	28,500	1985–2006
`comp.text.xml`	79,700	1995–2006
`microsoft.public.xml`	90,300	1995–2006
`comp.text.tex`	363,000	1981–2006
`XML-L`	8,342	1997–2006
`TEI-L`	9,407	1990–2006
Total	597,249	

Sources: null searches with date limits in *a*) Usenet newgroups: Google Groups; and *b*) Mailing lists: LISTSERV.

Overall the 5,313 messages retrieved[10] (section 3.3.3 on page 166) represent

[10]It is important to note that we use the total number of messages retrieved (representing all

0.95% of the total. However, the huge preponderance of messages on `comp.text.tex` skews this value heavily, because the majority of postings there have nothing at all to do with either data representation or document markup (most in fact relate to the use of TEX language or to mathematical formatting). If we exclude this newsgroup entirely, the percentage sample size is 2.46%. However, *some* messages on `comp.text.tex` clearly *do* concern editing structured documents (the very ones retrieved), so the effective percentage sample size is lower, perhaps around 2%.

3.4.2.2 Procedure

The survey was piloted for a week in April 2006 and the revised version was made available online between February and April 2008 using the *phpESP* web survey package. All respondents were guaranteed anonymity, but were provided with the opportunity to supply an email address if they were interested in a copy of the results.

The data was downloaded from the survey web site as a Comma-Separated Values (CSV) file, a plaintext spreadsheet export format. This contained the responses as typed or selected, but in their complete textual (uncoded) form. While this is visually self-explanatory to the human eye, it does not easily lend itself to direct manipulation in a statistical package. The data was therefore encoded manually using the *OpenOffice* spreadsheet. The multiple-choice responses were encoded by inspection into separate columns for each question, one column per response, using 1/0 codings to indicate selected or not-selected.

In order to facilitate the transfer of the data into a statistics package, these new columns were named by abbreviation in two parts (rows one and two of the spreadsheet: see the example in Figure 3.16 on the following page). The first represented the question and the second the response, so that for 'Operating System' (Q.3), with three possible answers, the upper part was `os` for all three, and the lower part was `win`, `mac`, and `unix` respectively. The effect is that row one contained name values only where we had created a new encoded variable (otherwise it contained the question number); and row two contained the second part of each of those new variable names, interspersed with the original survey

---

the readers who contributed to the answering of the questions), not just the much smaller number of originators who first asked the questions (and may themselves never subsequently have responded).

field names. (An exception was Q.5, the length of experience, where the response was in years; this was retained as a separate integer variable and recoded to a range later (see below).

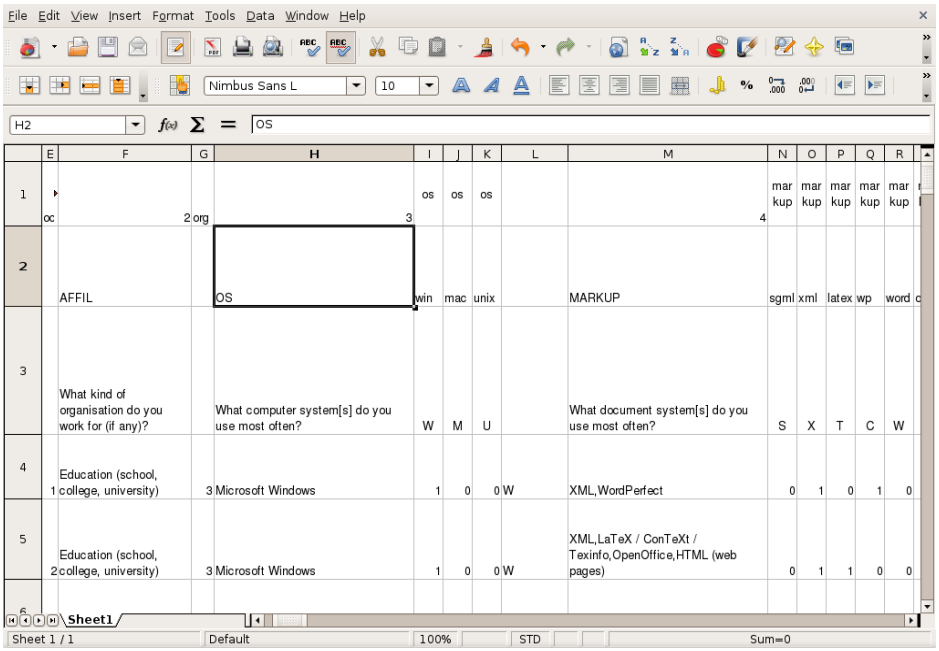


Figure 3.16: Encoding the data

The encoded data was exported to a TAB-separated file to avoid responses containing commas and semicolons interfering with their use as delimiters. An *awk*(1) script (appendix section C.2.1 on p. 392) was written to concatenate the newly-constructed variable names only where they occupied a position in row one; this presence or absence was used as a signal for whether the data occurring in that column should be written out or not, enabling the program to distinguish between the newly-encoded variables (which were needed for the analysis) and the original textual survey data (which was not).

The output was formatted by this script as a control file for data entry into the P-Stat analysis package (Buhler & Buhler, 1990) (see appendix section C.2.2 on p. 394). When processed, this constructed the initial dataset. This was modified with a short set of commands (appendix section C.2.3 on p. 395) to perform some further encoding, to check the total number of responses for each question, and to rearrange the logical order of the response variables, which had been deliberately ordered differently for purposes of control, as described in section 3.4.1 on page 176.

The cleaned dataset was then used to produce the tables shown in section 3.4.3 on the next page, using the `SURVEY` command and (later) the `TEXT.WRITER`

command to generate LaTeX typeset tables. Subsequent querying and ad-hoc retrieval was done manually.

3.4.3 Analysis and results

The small sample size (62) noted in section 3.4.2.1 on page 185 reflects the narrow band of interest available for analysis. We have maintained throughout that the target population for adaptations affecting the usability of editing software for structured documents consists of the non-experts in markup. Whatever walk of life they come from, they are therefore by definition *not* to be found in the ranks of markup experts, professional documenters, or technical writers, who have extensive experience and training in the application of markup, whether visual or logical. Unfortunately, there is no known representative body of the uncountable number of business, administrative, governmental, academic, and personal authors who create structured documents. This may in itself be an indication of the lack of awareness engendered by existing software which is 'just acceptable enough' to be tolerated; or perhaps of the wider problem that document structure is not taught in professional education except in specialist areas.

3.4.3.1 Q1–Q7: Background variables

The responses for Q5 Experience (years) showed a heavy concentration on a few values. As a result, these were collapsed into two groups (up to and over 15 years). The Q3 Operating System responses represented the three major categories (no 'Other'). The distribution of these responses against Q2 Organisation and Q1 Employment is shown in the composite chart in Figure 3.17 on the following page. In this chart, each operating system icon represents one respondent, spread left and right against the vertical line dividing the two age groups.

Just over 70% of respondents were in educational institutions (44 students, academics, and administrators). The poor response from business and government users was disappointing, but may be due to lower levels of interest in structured document handling, especially in areas where there is little or no awareness of the concept.

Over 75% of respondents (47) had 16 years or more experience of working with

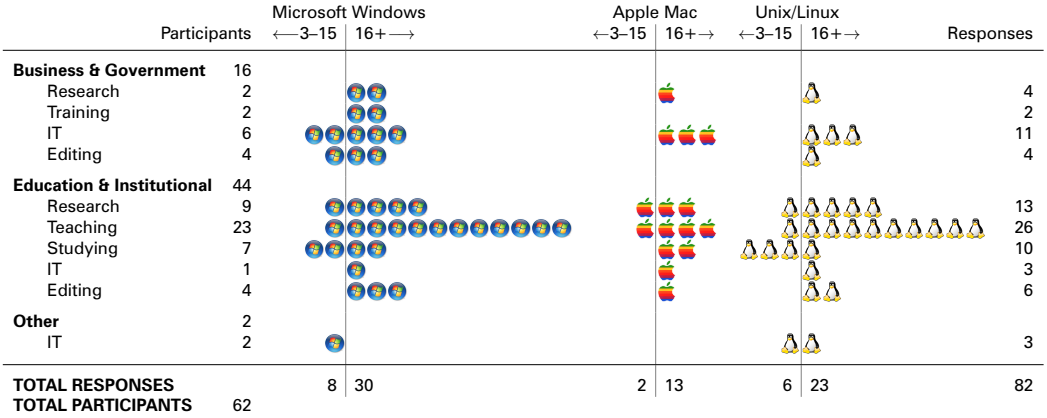


Figure 3.17: User survey: distribution of responses by Experience and use of Operating System against Organisation and Employment

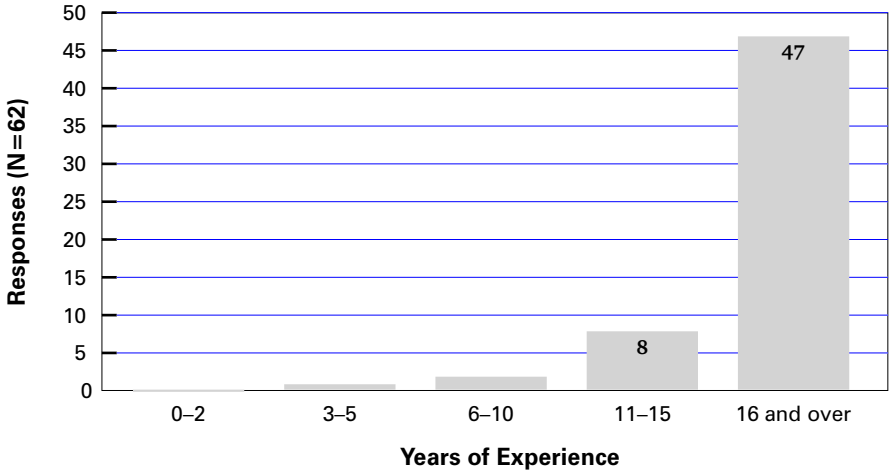


Figure 3.18: User Survey: Responses by Years of Experience

documents on computers (see Figure 3.18). However, they had experience with multiple computer systems, as those 47 respondents accounted for 66 out of 82 [multiple] responses (80%) to Q3 Operating System (Figure 3.17).

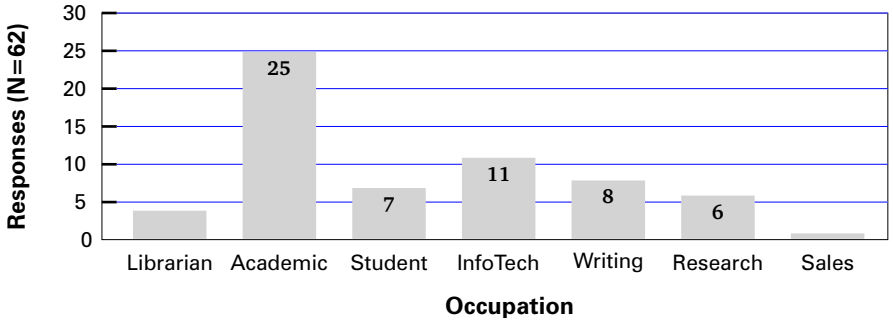


Figure 3.19: User Survey: Responses by Occupation

Overall, the background categories were too sparsely filled for further statistical purposes, and were collapsed for the subsequent analysis. For Q2 Organisation,

the education, student, and research categories became one category, 'Education/Research'; and the business categories and government formed another, 'Business/Government'; with the remaining two combined as 'Other'.

Occupation was more evenly distributed, but given the high percentage of educational employers, it was unsurprising that 40% of respondents held academic positions (Figure 3.19 on the facing page). Values for this variable were therefore also collapsed, but less aggressively: the responses under Library and Marketing were added to the Research category.

As we have mentioned, the low priority accorded to document structure in existing software as well as professional education may account for the skew in experience, background, and employment. However, if this holds true, the sample may be a fair, if small, representation of the population we are addressing.

XML (38) and HTML (35) between them accounted for 44% of usage of markup systems (Figure 3.20); LaTeX (24), *Word* (24), and *OpenOffice* (21) were roughly equal at 13–14% each. Wiki experience (14) was surprisingly high at 8% but all others (including SGML) were at 2% or below. From the comments made in later questions, it was evident that *Word* and *OpenOffice* were used as givens (that is, documents were supplied in that format), regardless of whatever conversion or subsequent processing was done.

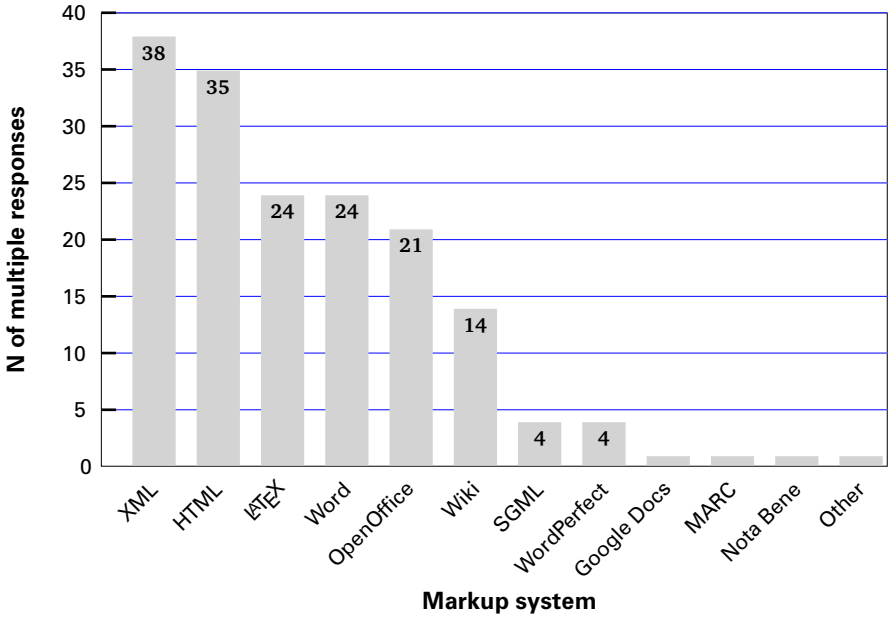


Figure 3.20: User Survey: Q.4 What document [markup] system[s] do you use most often?

Most respondents worked principally with text documents (48: 77%); other

document classes were in the single digit percentages. This was found to be useful: for the purposes of this investigation it was important that the principal concern was 'document' rather than 'data' usage (see section 1.1.3.3 on page 22).

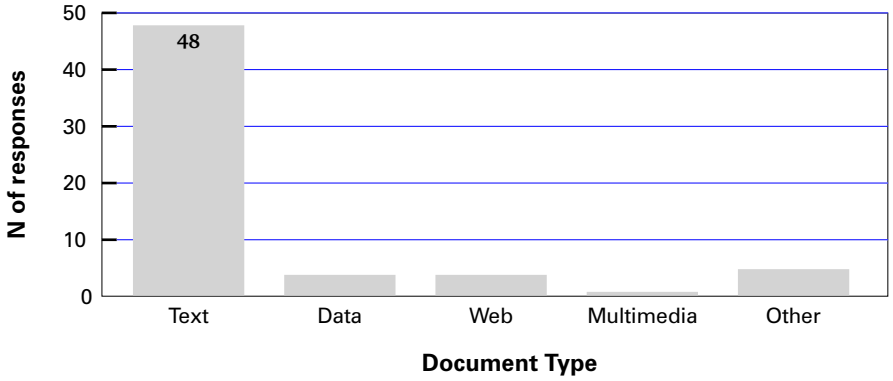


Figure 3.21: User Survey: Q.6 What types of document[s] have you had most experience with?

The editors most respondents had experience of were *oXygen* (20: 24%) followed by *Word* (11: 13%), *Emacs* (9: 11%), and *vi* (6: 7%) (both these last two were used for both XML and LaTeX). Other LaTeX editors accounted for another 12%. The *Arbortext* editor and *OpenOffice* rated 5 (6%), but there was a very long tail of other products (Figure 3.22).

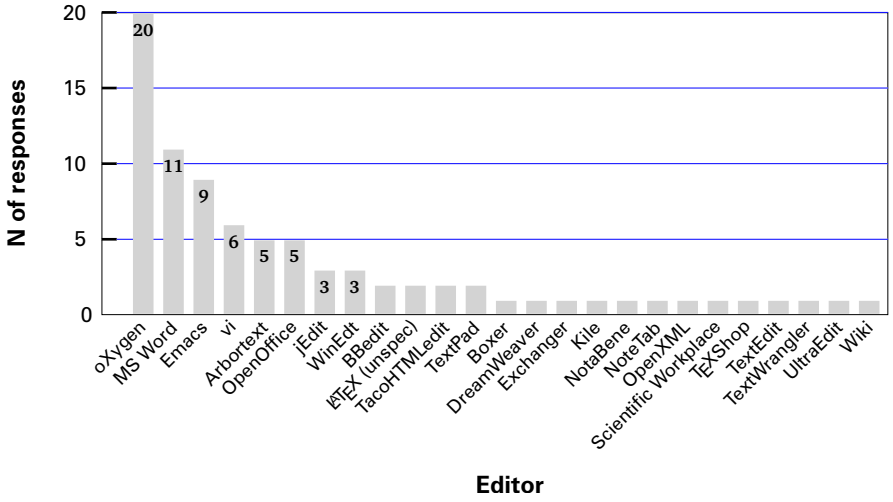


Figure 3.22: User Survey: Q.7 What editing software do you prefer to use for structured documents?

Many of the editors mentioned in the tail were either specialist products for a vertical market or general-purposes editors being used for editing structured documents. This bears out the anecdotal evidence mentioned in section 3.2 on page 133 that non-structured editors are frequently used even when there are structured products available.

3.4.3.2 How do you create a new document of the right type?

Three methods clearly predominate here (Figure 3.23): editing an empty file to specify the document type (32: 31%); reusing an existing document [whether of the relevant type or not] (30: 29%); and using the New document toolbar button of menu entry (29: 28%). However, six responses indicated that their menus did not contain any entries for the types of document they wanted; four used the File|New (wordprocessor-type) menu entry; and two said they could create new document types by compiling the DTD/Schema.

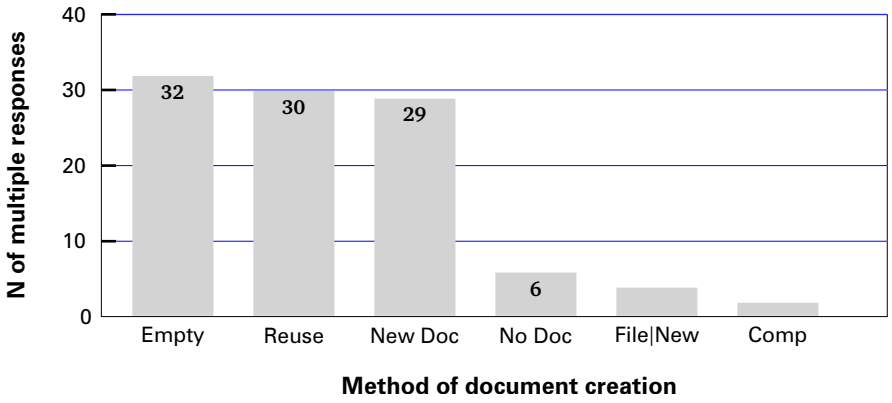


Figure 3.23: User Survey: Q.8 How do you create a new document of the right type?

3.4.3.3 How do you give the title, author, and other key information for a new document?

The two top methods were: typing the metadata as the markup is inserted (27: 34%) and filling in the relevant blanks (content of elements) when the document is created (22: 28%). The first would typically be used in manual and semi-manual editors, where the markup is either typed in manually or added via a menu (but not added automatically); the second is more common when using tightly-specified DTDs/Schemas where the metadata elements are added automatically at document creation time, and result in affordances labelled with the type of metadata expected (for example, 'Title', 'Author', etc).

The third method (Visual: 11: 14%) applies to wordprocessors, where visual formatting is the only way to identify information; followed by Automatic, where the metadata is predetermined by the business process, and is therefore inserted automatically at start-up.

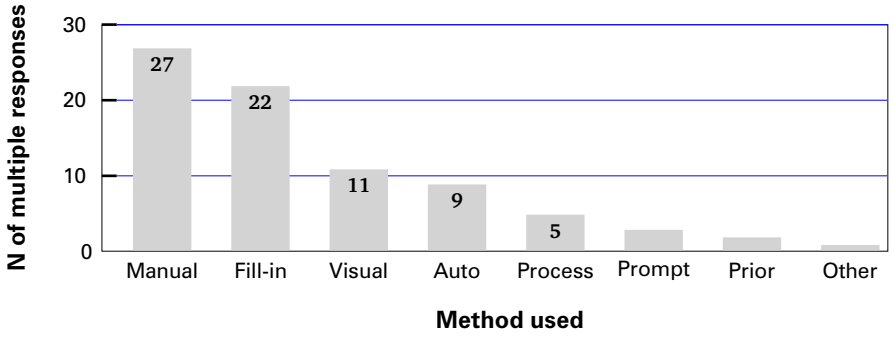


Figure 3.24: User Survey: Q.9 How do you give the title, author, and other key information for a new document?

Less common were the Defined Process (5: 6%), where the document creation is part of a wider process; Prompted (3: 4%) where the editor itself is triggered by the DTD/Schema to ask for the metadata (similar to the fill-in-the-blanks method, but prompted for in a separate window); Prior (2: 3%), which is similar to Automatic, but with the metadata entered by the author; and Other (1: 1%).

3.4.3.4 How do you tell your editor to start a new section (or chapter, subsection, subsubsection, etc)?

The leader by a long margin was the Manual method (45: 47%), typing the markup and text together; although the respondents used a wide variety of editors, headed *oXygen* with 17 and *Word* with eight.

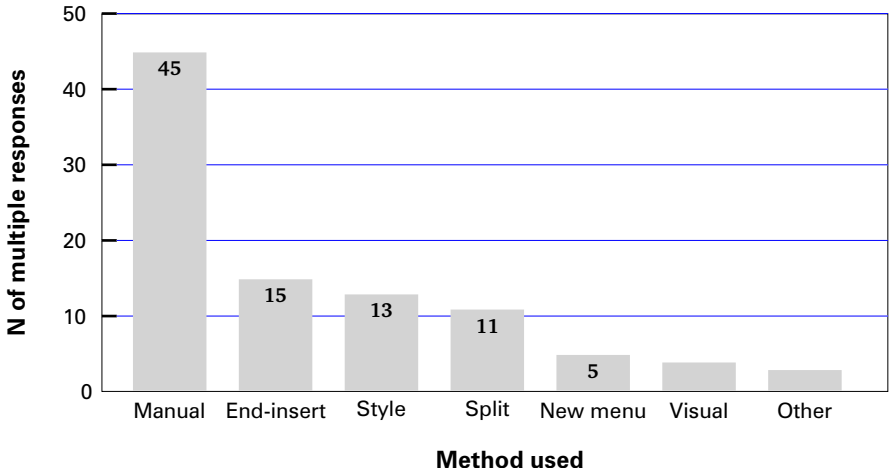


Figure 3.25: User Survey: Q.10 How do you tell your editor to start a new section?

Moving to the end of the current section and using the Insert function rated only 15 (16%), despite being the 'expected' method (the one customarily demonstrated and taught as 'the way' to insert a new element in element content.

This was closely followed by two contrasting methods: the use of styles from a template (15: 14%; *Word* and *OpenOffice* predominated here with 5 apiece); and the method of splitting the current sectional element at its end to create a new one (11: 11%). The remaining methods rated five or less (using a New Section menu; using manually-applied visual styling; and 'Other').

3.4.3.5 How do you apply formatting or styling?

Despite the heavy emphasis on manually-inserted markup in the foregoing questions, respondents placed automatic styling and formatting at the top (40: 47%), implying that whatever method was used to create the markup, they were relying on a stylesheet to ensure that its appearance was automatically correct.

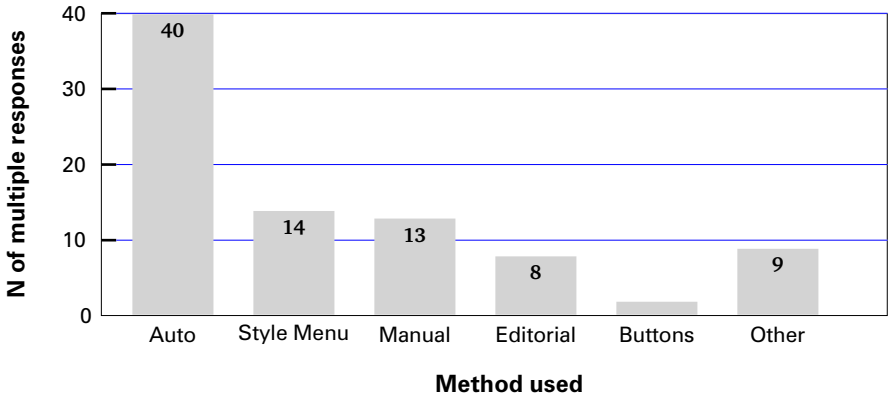


Figure 3.26: User Survey: Q.11 How do you apply formatting or styling?

Much lower came the use of the Style Menu (14: 16%) with predefined styles, but not automatically bound to element markup; and manual styling using the font and size menus and the B/I/U toolbar buttons (13: 15%). Only eight respondents relied on an editorial team to do formatting *post hoc* (9%), and only two used predefined style buttons (2%). However, the 'Other' category gathered nine responses (10%), mostly single occurrences of editors, but including four *oXygen* users, indicating that there are other methods in use.

3.4.3.6 How do you move blocks of text around when you edit a document?

Almost all respondents use the cursor-driven mark–cut–paste method (58: 71%), which is fast and effective when the entire block fits within the current window, and the block does not contain markup (such as lists or sections) which might conflict with the structure if moved to a location where they are not valid.

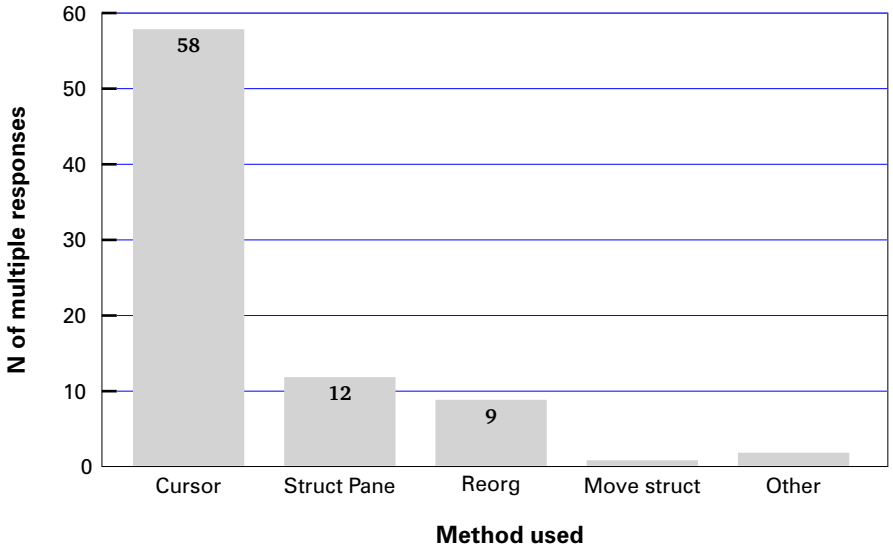


Figure 3.27: User Survey: Q.12 How do you move blocks of text around when you edit a document?

The only two other substantial mentions were for the use of a structure pane where much larger sections can be marked and moved (12: 15%); and (subsumed across other responses) the need to reorganise the markup (promoting or demoting) after a move (9: 11%).

3.4.3.7 How do you navigate around the document when editing?

Most respondents used a combination of searching (50: 30%) and scrolling (49: 30%) to move through their documents, and this pattern was evenly distributed across almost all editors used.

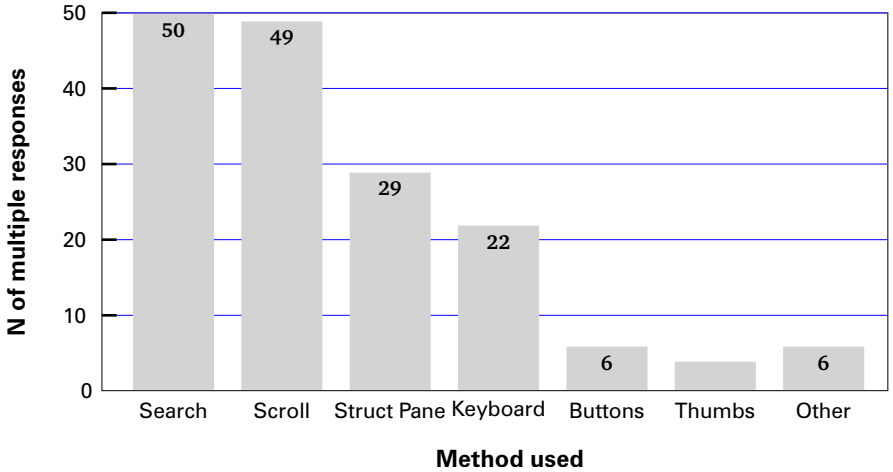


Figure 3.28: User Survey: Q.13 How do you navigate around the document when editing?

However, 29 of them (17%) used the structure pane in conjunction with the two primary methods, and 22 of them (13%) used the keyboard shortcuts (PgUp, PgDn, and their markup-based equivalents) to cycle or jump backwards and forwards through the document structure. The use of dedicated navigation buttons was low (6: 4%; of limited availability in any case); and the use of page-preview thumbnail images (4: 2%) was also low — a little surprisingly, given their easy availability in most PDF preview applications. The 'Other' category contained six unidentified methods (2%).

3.4.3.8 How do you add blocks like tables, figures, lists, sidebars, etc?

Manual markup insertion (typing) again predominates here (34: 35%), with 22 respondents using only this method. However, the use of toolbar buttons to insert these classes of element types is substantial (25: 26%), implying that they do provide the intended affordance. Similarly, the use of the Insert menu is also comparable (21: 21%), and over half of the users of these two methods use both of them (by contrast, only six of them also use the manual method above).

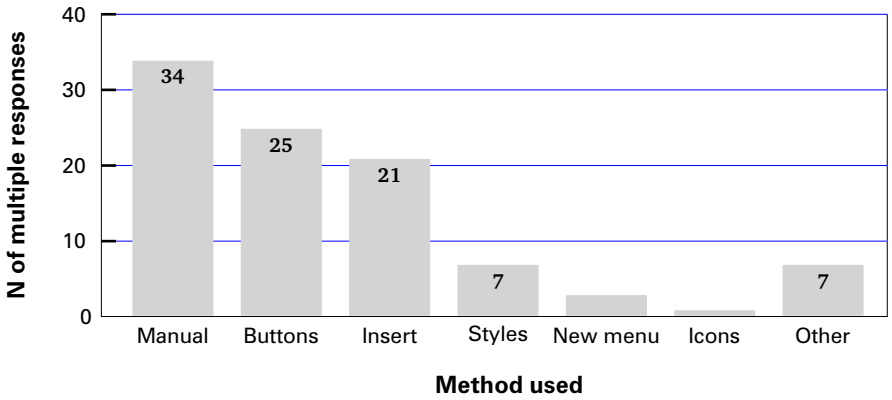


Figure 3.29: User Survey: Q.14 How do you add blocks like tables, figures, lists, sidebars, etc?

Use of the Styles menu is common among users of wordprocessors (7: 7%). A control method (the New menu function, which is not known to exist in any product) was claimed by three respondents, which appears to be a misunderstanding of the products represented (*oXygen* and *Word*).

3.4.3.9 How do you create or edit linking items like cross-references, footnotes, bibliographic citations, hyperlinks, acronym references, etc?

Manual insertion of links by typing the markup was the most common method (33: 35%), well ahead of the use of toolbar buttons or keyboard shortcuts (22: 23%) and the use of menus and dialogs (15: 16%). This would seem to bear out the contention made in section 3.2.6.3 on page 155 that editor support for linking and cross-referencing is at best underdeveloped.

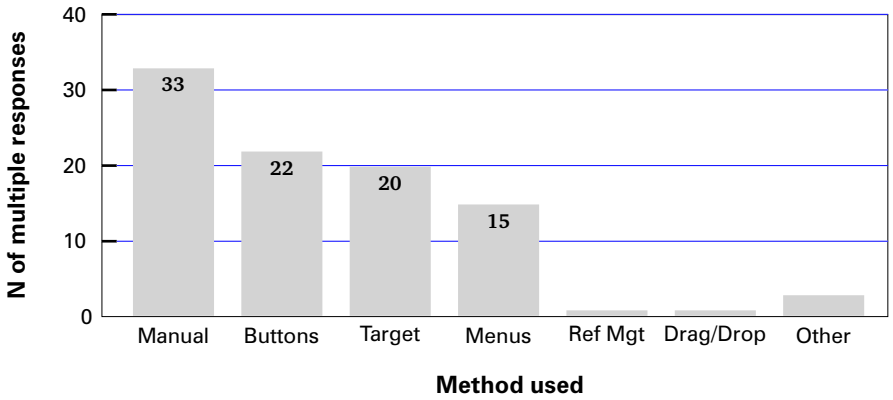


Figure 3.30: User Survey: Q.15 How do you create or edit linking items?

In 20 responses, the respondents said they had to create and identify the target of the link first, and then use one of the other methods to create the link (21%); only three of those 20 failed to indicate what other method was used. The use of reference management software and the alternative of drag-and-drop (both commonplace in wordprocessing) was virtually non-existent with one mention each.

There appears to be no reference management system that is capable of pushing links to an XML document, possibly because the element names vary per DTD (although support for XHTML, DocBook, and TEI should be straightforward) and because the cited references would need to be included in the document in order for the ID/IDREF link to resolve; or to be held elsewhere and use xml:link.

3.4.3.10 How do you know what your document-in-progress will look like?

Great reliance was placed on the structure of the document being right, and the appearance following this correctly (36: 27%), but only seven of these responses mentioned this method alone. All the other methods mentioned were in the

15–19 range (11–14%): reliance on a separate editorial team for the appearance was a minority response (9: 7%).

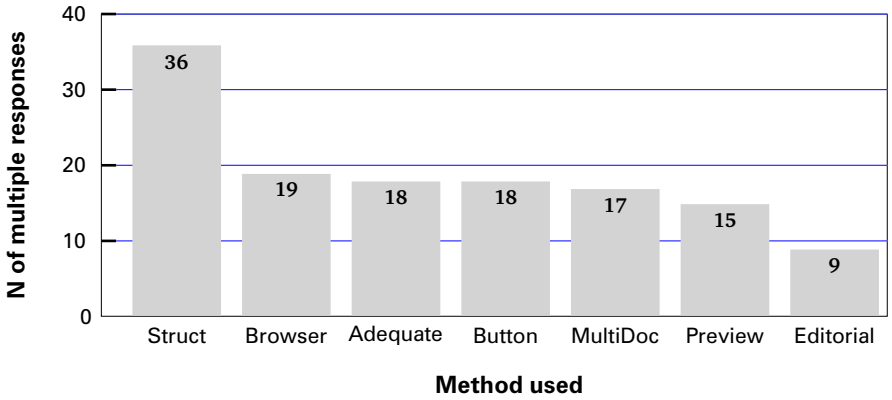


Figure 3.31: User Survey: Q.16 How do you know what your document-in-progress will look like?

The method of previewing in a browser window, reloading the page when required, got 19 mentions (14%), but 18 respondents (14%) said the interface provided by their editor (WYSIWYG or not) was adequate, although eight of those 18 also indicated they used the browser preview method.

The view or preview was initiated by a toolbar button or keyboard shortcut by 18 respondents, indicating that it was not an automatic function, and therefore probably not claimed as a WYSIWYG editor. Similarly, 15 respondents (11%) said the formatted output was in a separate typeset window, but only four of those were among the 18 who said they brought the display up separately, from which we conclude that at least eleven respondents (9%) were using systems with a separate preview window which *did* appear automatically.

3.4.3.11 What best describes your general approach to creating and maintaining structured documents?

In their view of their own approach, 35 respondents (38%) supported a Descriptive Markup approach (using editing software that lets them describe and identify the appropriate structure themselves). A third (12) of those 35 chose no other option, but a further 12 also chose the second-largest category, confirming that they did indeed 'use editing software that encourages and supports structure' (34: 37% — but of those 34, an entirely separate 14 chose no other option).

The other method, Prescriptive Markup (where the DTD/Schema prescribes and rigidly enforces a limited pattern), was indicated by 19 respondents (20%) but

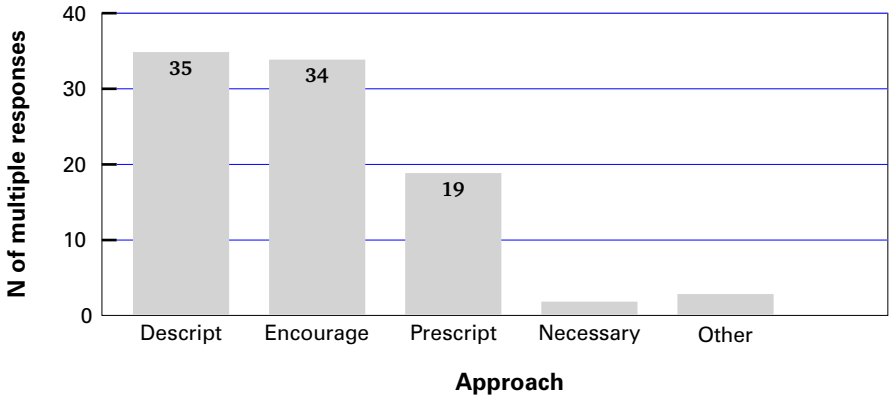


Figure 3.32: User Survey: Q.17 What best describes your general approach to creating and maintaining structured documents?

seven of these also used Descriptive Markup. Two respondents said they only used structural markup 'where necessary'.

3.4.3.12 When advising others (for example, colleagues, clients, students/trainees, friends) about software for editing structured documents, what specific feature(s) do you suggest they look for, or guard against (if any)?

This question provided an opportunity for respondents to provide some of the decision points they would suggest in choosing an editor. The matrix of combinations of the multiple responses is too sparse to make any judgments on combinations of reasons, but the totals are shown in Figure 3.33 on the next page.

The need for compliance with prevailing standards (XML or LaTeX) was clearly paramount (17: 15%), possibly indicating that this (as an exogenous constraint) is more important than the desire for structural control (4: 3%) in itself. But the need to be able to customise the software (8: 7%) and integrate it with other business processes (7: 6%), particularly LaTeX (6: 5%) is perhaps an argument for the need for flexibility in both markup systems.

Five responses (4%) claimed there were no specific suggestions, but the same number emphasised a need to avoid the attractions of WYSIWYG and wordprocessor interfaces, and to go for simplicity. However, a similarly low number (4: 3%) emphasised the need for WYSIWYG, which contrasts with the findings in the Requests Analysis (section 3.3.5 on page 168), where WYSIWYG was a prime requirement.

A need for for Regular Expressions (4: 3%), keyboard shortcuts, macros, and a

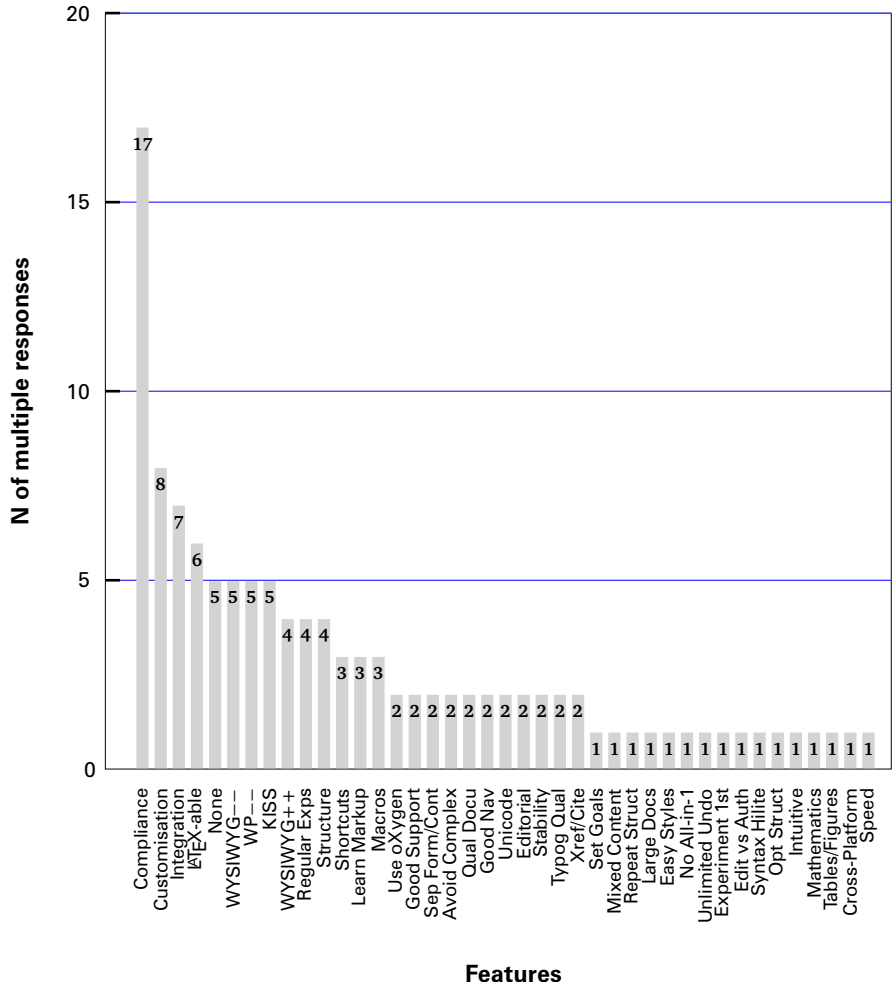


Figure 3.33: User Survey: Q.18 What specific feature(s) do you suggest others look for, or guard against?

knowledge of markup (all 3: 3%) starts a long tail of specific dos and donts.

3.4.3.13 What is the single most useful feature of the editing software that you use the most? (and what editor is that?)

A similarly large variety of responses is seen in this question (Figure 3.34 on the following page). Keyboard shortcuts (9: 14%) and the use of Regular Expressions (7: 11%) are seen as the most useful features, followed by ease of integration (5: 8%) and the ability to validate documents (5: 8%). This last one is slightly puzzling, as this facility is integral to any use of the common large-scale document types like DocBook or TEI; and placing emphasis on it indicates that perhaps some users are using editors that only check well-formedness.

Similarly contexual awareness (4: 6%) means programmatic sensibility to the

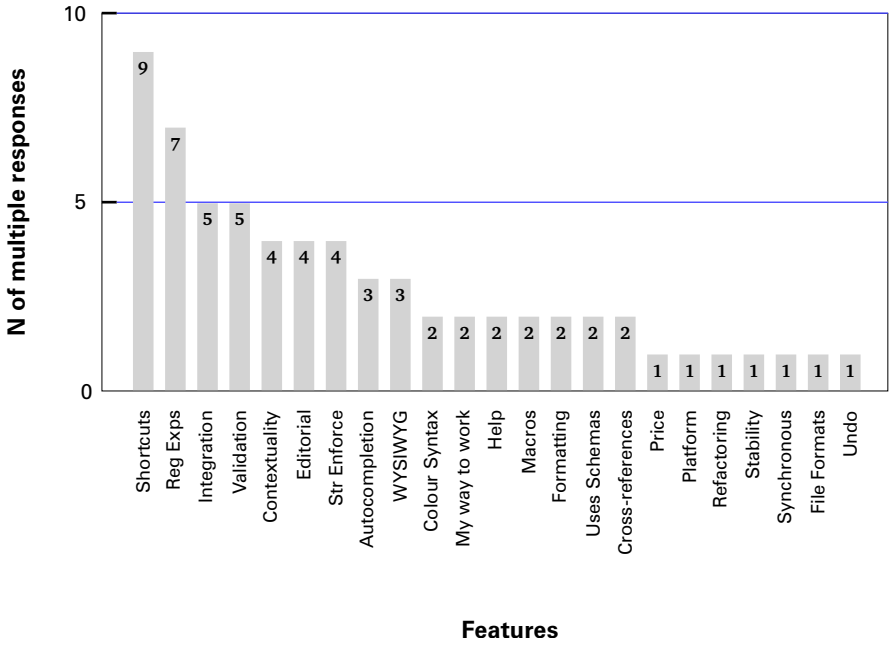


Figure 3.34: User Survey: Q.19 What is the single most useful feature of the editing software that you use the most?

constraints of the DTD/Schema, and would normally be taken as a given, as would structural enforcement of those constraints (4: 6%). Editorial functions, including spell-checking (4: 6%) are clearly a need for writers, as we saw in section 3.2.6.5 on page 156, but in this author's experience, have only recently become a standard part of the larger editors. Autocompletion (3: 5%) is a useful function, most notably found in *Emacs*, which enables element type names and token-list attribute values to be partially typed until they are minimally unique, whereupon the TAB key can be used to complete them without further typing. WYSIWYG makes it into this list but only with three mentions (5%).

3.4.3.14 What is the single worst feature of any editing software for structured documents that you have used?

At the other end of the scale, the 'worst' features (those poorly-implemented) contained some of the same ones as in the previous question, implying that the feature is important enough to be handled better. Element insertion (11: 20%) came first, although it was not clear from comments if this meant user dissatisfaction with the concept of an Insert function, or merely that it was done badly. Poor validation came second (6: 11%), either in the sense of non-conforming, or that the error messages were insufficient to identify the problem. Similarly, dissatisfaction with WYSIWYG (5: 9%) appears to relate to

poor-quality rendering rather than to the concept.

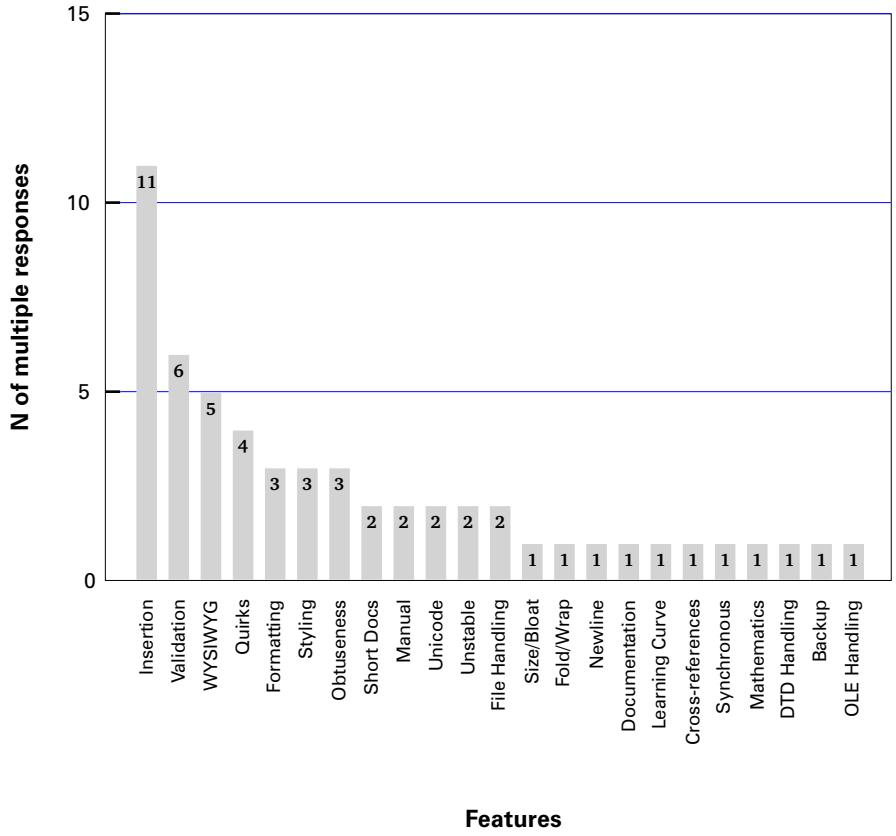


Figure 3.35: User Survey: Q.20 What is the single worst feature of any editing software for structured documents that you have used?

Quirky behaviour in the interface (4: 7%) tends to be either a relic of older interfaces which do not follow current expectations (for example, *Emacs*' copy–cut–paste shortcut keys, which predate the Ctrl-C/Ctrl-X/Ctrl-V keys popularised by Windows), or they are an attempt at innovation, or (worse) novelty-hunting. Poor formatting and poor styling rated 3 (5%), as did a quality we have named 'Obtuseness', by which the editor either refuses to perform a function, or performs it in an entirely unexpected manner.

3.4.3.15 When using an editing system for structured documents, have you ever failed to find out how to do something that the product was actually capable of doing? Please describe the circumstances briefly.

Locating something in the documentation was the primary failure seen by respondents (11: 35%). While computer documentation is anecdotally notorious, it would seem even less excusable that a system intended for writing documents

should itself be poorly-documented. Actually locating a feature was second (8: 26%), meaning it was not visible in the menus, toolbars, or command-set.

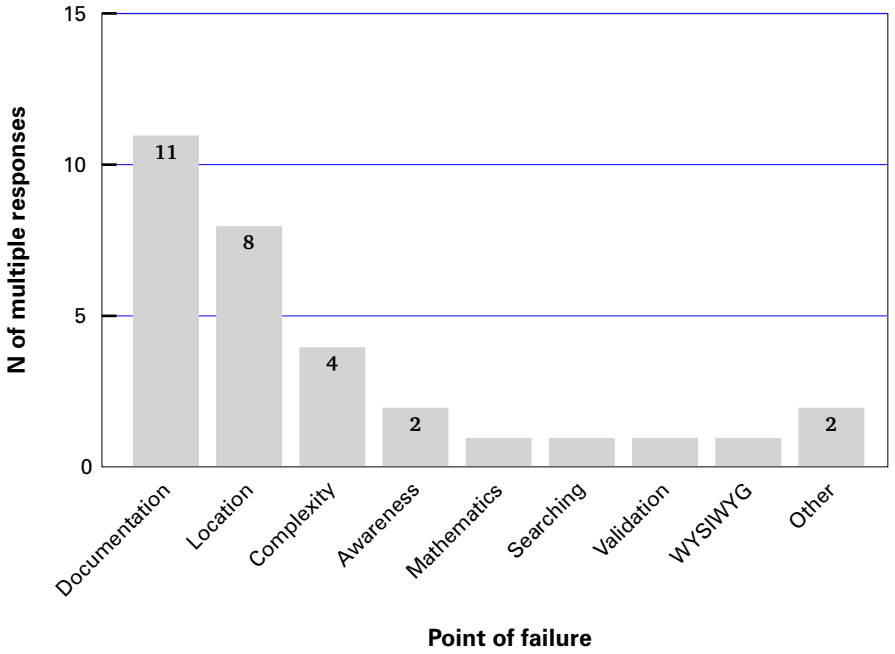


Figure 3.36: User Survey: Q.21 When using an editing system for structured documents, have you ever failed to find out how to do something that the product was actually capable of doing?

Complexity of the interface (4: 13%), while not specifically a non-feature, was mentioned in extenuation for failure. Lower-rated responses covered a user lack of awareness of features (2: 6%), and difficulties with mathematics, searching, validation, and the use of WYSIWYG.

3.4.3.16 What features would you like to see in an editing system for structured documents that aren't in any of them at the moment? (ie your WishList)

This section was relatively short: there were only 22 responses, and it must be assumed either that editors do actually provide the majority of features required (albeit perhaps suboptimally), or that the sampled users were not aware or imaginative enough of the possibilities. Documentation was again highly placed (6: 20%), as was the need for WYSIWYG (5: 17%), despite earlier rankings to the contrary.

'The interface' (unspecified) was one concern (4: 13%); more specifically the quality or usability of menus (3: 10%), Regular Expressions (2: 7%), and the availability of Styles (2: 7%) in the manner of a wordprocessor.



Figure 3.37: User Survey: Q.22 What features would you like to see in an editing system for structured documents that aren't in any of them at the moment?

3.5 Conclusions

It was clear from the many responses in various forms that the overwhelming requirement is for a WYSIWYG interface, perhaps even when this conflicts with the requirements of the document type in terms of markup validity. The presumption appears to be that the software should deal with this itself, and that it should not be a preoccupation of the author.

Many of the other topics raised in these studies follow on from a decision or requirement to use a WYSIWYG interface because they require the application of formatting implied by the markup. That is, when markup is applied that identifies a specific type of information (such as a new section, or a word in another language), the formatting is the *only* way of showing that something has happened, because the markup itself is not on view.

This was foreshadowed in the Expert Survey in the comments that software is often selected on the basis of what the user finds familiar (see the first item on page 131). Assuming the WYSIWYG model, therefore, means that we need to take care that the testing adequately takes into account the underlying nature of the markup. In a non-WYSIWYG editor, where the markup is visible, it is relatively straightforward to see and understand what is happening when a writer invokes a particular function such as creating a new section or identifying a foreign word. Even in a non-XML or non-LaTeX environment, the implicit markup such as a double blank line or centred text in capitals is evidence that something other than normal text is being typed. The behaviour expected by the writer in both WYSIWYG and non-WYSIWYG modes must therefore form a part of our analysis in the next chapter.

In going from the views of the experts, via the analyses of the software and user requests, to the requirements of the users, we have therefore gathered a set of topics for analysis, from which we must separate out what we need to test, with a view to identifying some palliative measures and evaluating their usability and acceptability, within the limitations we imposed in section 1.2.2 on page 43.

It is unlikely that changing the method of working relating to any of these topics *in isolation* would be sufficient to attract or hold more authors to a particular structured-document editor. However, their incorporation in an interface sufficiently resembling the conventional wordprocessor, together with enough adaptation of the behaviour to avoid the rejection due to unexpected results, can at least now be tested in properly controlled circumstances. This may indicate

which of the areas perform acceptably, if any, and which require further examination.

CHAPTER 4

Modelling and testing

1. WHAT HAVE WE MEASURED? — Principal features and functions.
2. METHODOLOGY FOR BUILDING AND TESTING — Personas revisited.
3. SEMANTICS OF THE SELECTED FUNCTIONS — Non-menu controls (keyboard shortcuts) — Creating and opening documents — Insertion of new material — Formatting controls — Block moves — Referencing — External insertion — Editorial assistance. 4. BUILDING THE MODEL. 5. TESTING — Test harness — Test schedule — Rubric — Expected responses — Administration.

> *A picture is often said to be worth a thousand words. Similarly, an interface is worth a thousand pictures.* (Schneidermann, 2003)

In Chapter 3 we investigated the requirements expressed by users of XML and LaTeX systems for structured documents, and analysed the practical application of the editing software. This identified a number of discrepancies between the requirements of the user and the provisions of current interfaces.

There may be many reasons for these discrepancies, including the differences between the mental models of what the system should do held by the programmer or system designer, and the models held by the writer or editor; the pressures of marketing policies of the vendors; and differences between the physical requirements of the products and the expectations of the users. However, our objective here is to identify ways in which the user requirements might be met by modifications to the interface, whatever the underlying causes of the differences.

In this chapter we identify what it is that we have measured in Chapter 3, and put together the assorted features and functions that users identified as problematic. The semantics of these features are analysed in section 4.3 so that we can capture precisely what the respondents told us they would expect to find if the interface 'did it right'; and with this discussion in mind, we proceed to building the model in section 4.4. In section 4.5 the testing approach is described. Each of these stages builds upon the preceding one, to try and ensure that we do not end up modelling or testing some aspect which is not warranted by observation and measurement.

4.1 What have we measured?

From the information collected in Chapter 3, we identified that these discrepancies made themselves evident in the presentation or behaviour of individual items in the interface (or their absence), where some key criteria were not being fulfilled, principally:

1. Functions required by the user were not being exposed in a usable manner — that is, the functions were present in the program, but were either not visible (not affordable), or were not easily reachable (not easy to grasp);
2. Functions were not presented in terms understood by the user — that is, the functions existed and were visible, but were called by a name or label that was not understood as that functionality to the user;
3. The behaviour of a function did not accord with the expectations of the user — that is, unexpected things would happen when a seemingly-obvious function was exercised;
4. Some functions were simply absent from some interfaces — that is, the program did not provide those functions at all.

It can readily be seen that a key component of each problem centres on the degree of affordance provided by the function. We have established in section 2.1.4 on page 66 that Gibsonian perception is taken as a prime requirement, but while the 'degree of affordance' implies an infinitely-variable scale, it is clear from the users' requirements in the preceding chapter that their reaction to the key criteria above is *binary*; that is, a function is either there or not there. The fact that it *might* be there (after some hunting), or that there is a

related function that might almost do the job, are degrees of refinement which are not relevant when a user's frustration with a program is growing.

The four enquiries (Expert questionnaire, Request analysis, Software analysis, and User questionnaire) revealed a large number of individual features and functions that were mentioned by respondents or identified by analysis, some with relatively low frequencies of occurrence or usage. The diversity of these results means that assembling a synthesis is restricted to requests or requirements that are frequently-occurring enough to be achievable and measurable. Some of the difficulties in deriving an implementation plan from such a synthesis are examined in section 4.4 on page 281.

The objective is to identify which of the features and functions discussed in this chapter can be redesigned or modified, and tested in an interface in order to enable or improve the usability of an editor for the non-markup-expert user. For each of the principal features listed below, we summarise the nature or extent of any changes that might be made to the present way of working. 'Principal features' in this context means the functions or features that were ranked or rated towards the top of the categories discussed in the four enquiries, subject to them actually being actions or behaviours that can actually be modified.

This means setting aside requirements that would require re-engineering large parts of the editor *other than* the interface, such as the facilities for macros or customisation; and those requirements which properly belong to the implementation of the a program as a whole, such as standards compliance, integration, stability and platform-independence; as well as those requirements which are optional in XML or LaTeX terms in editor design (that is, they do not affect the structural formation of the document), such as the inclusion of a facility for tracking and reviewing comments.[1] It also means excluding external factors such as the hierarchical filesystem — something which new users may be unfamiliar with, and which may cause them problems, but which are far outside the scope of this research.

In compiling this list from the disparate results mentioned, we supplemented them with a small number of parallel results from other recent research. The relatively low-frequency responses in all the available sources indicated that a form of triangulation would be useful, by means of which a position or status can

[1] Ironically, as we have seen in section 3.3.5 on page 174, this is one of the facilities which aids in perpetuating the use of wordprocessors in author-publisher interaction (see also section 6.4.2 on page 366).

be determined by reference to several 'sightings', each of which provides an indication of distance and direction.

However, in the absence of a suitable grid and empirical data attaching to the 'sightings', evaluating them involves making a judgment on how to balance the weight that survey respondents may have assigned to the functions against the technical effort that would be involved in implementing them. User requests for facilities outside the bounds of current technologies are not in scope for this research. In some cases the requirement is qualitative, not quantitative, because it was mentioned but not measured (for example, 'familiarity of the interface' in the Expert survey); we treat these as givens for any implementation. This form of triage is commonplace in decision-making, and is sometimes represented as balancing difficulty of completion (DOC) against requirement importance (RI).

While we have data on the importance of the functions from the foregoing surveys, in the form of their relative frequency of mention, the difficulty of implementation must therefore be taken into account, and this must inevitably be subjective, as we have no access to the programming techniques of all the software involved. However, we are not seeking an absolute means of distinction between functions but rather a relative one (that is, how much more or less difficult is one function to implement than another). In addition, some of the changes are known to be possible, as the data needed for implementation is already a part of the data model of the file format involved (XML or LaTeX). Some further comments on the implementation possibilities are in section 6.3 on page 341.

4.1.1 Principal features and functions

We list these in the order of importance derived from their ranking in the foregoing surveys and inquiries. When we come to examine them in relation to testing, we group them according to their implementation (section 4.3 on page 222).

WYSIWYG: This was the prime requirement in the Requests Analysis (Table 3.13 on page 169), the second highest in the Expert Survey question on usefulness (Figure 3.9 on page 122) and in the User Survey wishlist (Q.22, see Figure 3.37 on page 205), and the third 'worst feature' in the User Survey (Q.20, see Figure 3.35 on page 203).

While definitions vary, most non-experts appear to believe that an editor 'must be WYSIWYG', possibly because they are largely unaware that any other sort of interface exists. In the absence of any fundamental shift in the paradigms available, this means that we must regard it as a *sine qua non* for any system aimed at writers who are not markup experts. Wordprocessors in their default mode simply allow arbitrary formatting and little else; but for structured editing, the binding of structure to format is an integral part of the WYSIWYG paradigm, as evidenced in Figure 3.31 on page 199. This is notwithstanding the conceptual description of structured documents as instances of late binding (Reid, 1989) where the application of formatting and consequent removal of all traces of structural markup are delayed as long as possible.

Given this requirement, we gave no special consideration to the implementation of interface changes to *non*-WYSIWYG editors, as the operations required (renaming and relocation of interface components, especially menus and toolbar items) and changes to behaviour (such as repositioning of the cursor) are just as achievable in textual interfaces as in graphical ones.

As we explain in section 4.3 on page 222, no further action is taken in respect of WYSIWYG editing other than to state it as a pre-emptive requirement clearly required by the users.

Styling (formatting): The use of the Style menu ranked second in the User Survey question on how styling was applied (Q.22, see Figure 3.26 on page 195), and third in the methods use to create a new section (Q.10, see Figure 3.25 on page 194); it came top in the Expert Survey question on poorly-implemented features (Figure 3.10 on page 125) for failing to honour CSS, and third in the question on lack of facilities for inability to handle formatting (Figure 3.11 on page 127).

While changes to the internal styling mechanism of individual editors is far outside the scope of this research, there is scope for use of existing or new style-related affordances for the activation of some of the functions we discuss, particularly in connection with the detection of user intent, which we examine elsewhere (section 6.3.11.1 on page 356 and section 6.3.12 on page 357).

File Open: In opening and creating files, both Schema/DTD support as well as

DTDless editing require examination (support was first equal in the Expert Survey question on the lack of features in Figure 3.11 on page 127, and eighth in the Request Analysis in Table 3.13 on page 169; DTDless editing was fifth equal in the Expert Survey question on poorly-implemented features in Figure 3.10 on page 125, but is also related to the deduction of user intent, above). Schema/DTD support may also imply the use of XML Catalogs (third equal in the same question) and user awareness of the distinctions between descriptive and prescriptive approaches (first equal in Q.17 of the User Survey in Figure 3.32 on page 200). The ability to use a DTD or Schema is a requirement of any XML editor.

The use of the empty-file method *vs* the reuse of existing files method (items one and two in Q.8 of the User Survey in Figure 3.23 on page 193) may also be facilitated by changes to the interface, as may file conversion on import/export (first equal in the Expert Survey question on the lack of features in Figure 3.11 on page 127).

Block move: Editorial movement of blocks of text is principally done using the cursor (top mention in Q.12 of the User Survey in Figure 3.27 on page 196), with the tree view a poor second; from the designer's point of view this conflicts with the graphical paradigm implied by a WYSIWYG interface, but from the user's point of view it may indicate a conflict in the user's mind between what you are 'supposed to do' in the WYSIWYG environment (directly manipulate the text), and the "at arm's length" concept of manipulating one window in order to have an effect in another.

Promotion and demotion of ranked block-mode material (lists, subsections, etc) was top of the Experts' list of failures (Figure 3.12 on page 129) but rated much lower on the User Survey (Figure 3.27 on page 196).

Insert: The User Survey placed problems with the Insert function topmost (Q.20 of the User Survey in Figure 3.35 on page 203). We argue in the next chapter for the replacement of much of this functionality with the New function already mentioned, leaving the Insert function for inline markup (mixed content), special characters, line/page breaks, and inline graphics, where the semantics implied by the word 'insert' are well established.

Editorial: The use of the tree view is covered in 'Block move' in this list. There are markup-related problems with searching and spell-checking (item six in Q.19 of the User Survey in Figure 3.34 on page 202) and with the use of

Regular Expressions (item two in the same question). These functions are discussed further in Editorial Assistance (section 4.3.8 on page 279) but are not included in the tests.

Unicode: This was found problematic in the Requests Analysis (Table 3.13 on page 169), but as explained in section 4.3 on page 222, no further action is taken in the interface tests.

Keyboard Shortcuts: These rated first in the User Survey question on usefulness (Figure 3.34 on page 202), but their behaviour is unclear in the Software Analysis (Figure 3.14 on page 159).

- The Enter/Return key may perform an element split or a premature newline.
- The TAB character may be used for auto-completion (eighth in the User Survey question on usefulness in Figure 3.34 on page 202), but is redundant in the majority of cases (Figure 3.14 on page 159).
- Backspace and Delete erase text as expected in character data content, but their behaviour when they encounter a markup boundary is inconsistent (also in Figure 3.14 on page 159).
- Spaces are often [ab]used by users unaccustomed to markup and stylesheets in attempts to create horizontal white-space such as indentation.

Misuse or misunderstanding of these keys was mentioned by several of the experts and users questioned, and their importance is discussed in section 4.3.1 on page 226.

Metadata: The specification of metadata needs identifying in a machine-readable way from DTD or Schema sources (Q.9 of the User Survey in Figure 3.24 on page 194). In conjunction with 'File Open' in this list, the deletion of all document markup and content from a document should trigger the re-entry of metadata.

Sections: The creation of new sectional divisions is problematic in WYSIWYG editors when there is no markup displayed (top position in Figure 3.35 on page 203) because the author cannot be expected to know the details of element location, and a strict interpretation of the DTD/Schema may forbid a new division at the actual cursor location. The New menu is considered in testing as a possible solution.

Structural markup: A similar position obtains with the 'insertion' of tables, figures, lists, sidebars, and other elements in element content (third position in Figure 3.23 on page 193). The relative usage of manual insertion, buttons, and the Insert menu are discussed in Figure 3.29 on page 197.

Both this and the preceding item on sectional divisions may benefit from the techniques of SI (section 6.3.2 on page 344).

Navigation: Searching and scrolling take first and second place in the User Survey question on navigation (Figure 3.28 on page 197), whereas the use of a tree view comes only third (see also the Expert Survey results on useful features in Figure 3.9 on page 122); the difficulty may be caused by the limited amount of a large document that can be seen at any one time: an outline-mode or thumbnail presentation may be useful here.

Links and Referencing: The instantiation of ID/IDREF cross-references requires real-time dereferencing using styles and generated content: the User Survey question on this (Figure 3.30 on page 198) places manual methods first, buttons second, and the pre-allocation of the target third, with menu methods fourth.

Validity: Editors may display the realtime status of document validity, but there appears to be no method for user selection of the warning or override: this occupies second position in the User Survey question on poorly-implemented features (Q.20, in Figure 3.35 on page 203). The markup-inexpert user is unlikely to be in a position to fix partially-broken documents at import time. As explained in section 4.3 on page 222, no further action is taken on this item, as it must be presupposed in a WYSIWYG interface.

These 14 topics form the basis for the development of a set of variations to the toolbars, menus, and widgets in the traditional editing interface, which is constructed in this chapter. In constructing this model we will need to examine the semantics of keystrokes, toolbar buttons, and other objects in the interface in order to understand *why* a user will choose one over the other, or believe that pressing or clicking one will achieve a particular result in the face of evidence to the contrary (or no evidence at all), from which we derive the new model of the widget.

4.2 Methodology for building and testing

Measuring the effects of changing an interface requires a consistent procedure in constructing the changes as well as a consistent method of measurement. It also requires a static model of the existing interface which can be used as a benchmark to generate a baseline set of results.

It was apparent during the investigations that the software tested was quite disparate in form and appearance, although key parts of the interfaces broadly adhered to the prevailing paradigms:

- the menu layout in almost all cases followed the 'File–Edit–View–Insert–Format–Table…' pattern common to editing applications, although the submenus varied the deeper one went[2]
- ancillary control panes were placed to the left and right of the edit window as necessary, with multiple panes being stacked vertically;
- toolbar icons followed wordprocessor conventions for the common operations, but diverged significantly when it came to markup-specific operations such as split, join, insert, and remove.

It was also apparent that the model we needed was only of the interface itself, not of the underlying software which performs the document parsing, manipulation, storage, or formatting. All these latter are abundantly available in toolkit form, known as Application Programming Interfaces (APIs), in both FLOSS and commercial software fields (albeit not always in compatible languages or implementations), accessible to any programming team with the resources to undertake a project like the building of a new editor.

Interface toolkits are also freely available for the layout engines, editing surfaces, control interfaces, and typographical drivers, and are used in many such projects. However, the key element needed for building the model was not a tangible piece of software but a decision framework for which of the identified functions were amenable to testing in an interface. In common with most software projects, doing the actual implementation is a (relatively) straightforward — albeit very large — task; deciding *what* to implement, where, and how, is considerably less clear, by several orders of magnitude.

[2]At the time of first measurement, the 'ribbon' menu was not implemented in any of the products tested. See footnote 3 on p. 219.

It was entirely beyond the scope of this research to write an entire XML or LaTeX editor from scratch, even using the extensive toolkits available. It has taken over a decade for the existing XML systems (and even longer for LaTeX systems) to mature to a usable state for experts, even with the further decade or more background in SGML and TeX which many of the earlier vendors possessed. In addition, committing a model to executable code involves architectural actions and decisions which — even partially — prejudge the outcome. It can be extremely difficult to undo certain programming structures which have to be done in the first place in order to test a concept, despite the extensive techniques for parameterisation and indirection available in modern programming languages (Spinellis, 2007, p. 288).

However, there are several methodologies suitable for testing the interface aspects of program design and operation which do not require an executing program to be written, as we discussed in section 2.1.5 on page 70. These include flow-diagram walkthroughs, scenario presentations, paper prototyping, and questionnaires. Rubin (1994, 31–37) summarises the applicability and methodologies of the category he refers to as 'exploratory tests' and provides an example (coincidentally, of a wordprocessor interface). These are relatively low-cost options compared with writing an entire program, but in view of the widespread use of the prevailing interface, Microsoft *Word*, it would be desirable to complement the testing with some form of comparison as described by Rubin (1994), specifically, conducting a test in parallel (side-by-side) on *Word* and the experimental interface in order to confirm that the behaviour being tested is truly comparable.

As we have seen in the list beginning on page 212, some of the functions we have identified cannot easily be expressed in terms of diagrams or descriptive presentation, and the complexity of reason, choice, action, and interaction would require too many branches for a workable questionnaire. The decision was therefore taken to use Paper Protoyping, based on methodologies suggested in Snyder (2003), and using the facilities for such testing in the HFRG at the author's own institution. This is a rather more expensive option in time and effort, as it involves constructing a fully-detailed and realistic simulacrum of the interface, albeit in static form, for every eventuality likely to arise during testing; and the tests themselves are of necessity one-on-one events. As we saw in section 4.2, a large amount of the model concerns the interface alone, the realism of which is a classic Paper Prototyping problem (Snyder, 2003, p. 4).

Inclusion of a parallel test to validate against *Word* behaviour in these circumstances this would double the entire size of the testing task, including the time and the demands on participant testers. In the specific case of *Word*, the provision of affordances and their use by authors is already known from observation and experience, so a parallel test was felt to be unnecessary. However, to act as a check on this, a separate pilot test was conducted with the participation of two professional trainers in *Word*, and their comments used to confirm that the test represented actions and expectations that *Word* users would be familiar with.

Devising the specific tasks of the current investigation (the functions to test) necessarily concerned the fine detail of interface presentation and the users' interaction with menus, keystrokes, mouse clicks, and toolbars. As we saw in section 2.3.4 on page 98, there is a considerable body of research on the affordances and effectivities of interface widget selection. However, implementing these with techniques such as wireframing, or writing executable prototype programs, is very much more expensive than the options outlined above. In addition, there is relatively little research on the application of widget selection and placement to editing structured documents, and even less on the components of documents themselves. Indeed, provided that action x achieves the expected and desired result X in time t, it is unimportant to the functionality whether action x is a second-level menu item (click–move–release) or a drop-down selection (equally click–move–release). What does have specific relevance to the design of the interface is the placement and labelling of such widgets, and this was one of the major reasons for revisiting the software analysis in 2009 to measure that attribute of the functions which we termed **reachability** (section 3.2.6 on page 149).[3]

[3] A further consideration was the introduction of new control surfaces in Microsoft Office 12 (2007–2008) and 14 (2010), and in Windows 8. These replaced the conventional File–Edit–View–Insert–Format–Table… menus with a **ribbon** intended to facilitate those actions required most often by most users. This was initially restricted to *Word* in wordprocessor mode (ie, not in the XML mode which is a separate part of the post-2003 product). Whatever its merits compared with the traditional menus, it is of no immediate relevance here, where we are dealing with a more rigorous set of requirements than found in conventional office wordprocessing. We have therefore not attempted any synthesis of the 'ribbon', as the traditional menu interface still persists in almost all other document-editing applications.

4.2.1 Personas revisited

In section 1.3.3 on page 47 we sketched four personas who seemed to represent important classes of candidate for the use of a structured-document editor. In the light of the data collected in Chapter 3, especially in the User Survey (section 3.4 on page 176), it was felt important to review and possibly update the personas to ensure that they represented the perceived target audience[s]. The four personas as they were conceived before the investigative work began are shown in Figure 1.14 on page 50.

Table 4.1: Functions ranked against personas for perceived applicability to requirements

Function	Annie	Bruce	Carrie	David	Location discussed
Keyboard shortcuts	5	3	1	4	p 226
New documents	4	2	5	3	p 244
New material	3	3	4	4	p 246
Formatting	3	5	2	2	p 253
Block editing	2	1	2	3	p 260
Referencing	1	5	2	1	p 269

1 = unimportant; 5 = very important.

In revising the personas, the objective was to provide characterisations which would have more relevance to the specific functions we wished to test: before the investigations, the originals were accurate but generic. To do this, a table was constructed, showing the function groups down the side and the personas across the top, and each function group was ranked from 1 to 5 for their perceived applicability to each persona's requirements (Table 4.1).

During the analysis of the functions carried out in Chapter 3, the requirement importance of each group to each persona formed a part of the qualitative evaluation which determined whether and in what form a function should be included in the testing.

Annie: works for a company that handles the CVs of very highly-qualified people, running the web site and preparing reports on their backgrounds and experience for potential employers. She is a historian by training but also a computer scientist, so she understands the concept of markup and the need for a robust structure. She's under pressure to get each job done, but the information is so widely-scattered and inconsistent that there is a lot of copy-and-paste as well as original writing.

Currently she uses *Word* with a stylesheet, because there is extensive document repurposing. She regards the provision of a familiar interface as essential.

Reports to potential employers need very consistent identity and formatting to prevent bias; her stylesheet provides some of this, but the copy-and-paste keeps messing it up. Reports are needed in different formats for different clients, so switching stylesheets has to be a very robust operation.

Bruce: has his own publishing business producing specialist reports and journals in a variety of fields; mostly, but not exclusively, scientific. He arranges for experts to write articles, then edits and formats them himself for the printer and web site (his colleagues deal with sales, marketing, and finance). The incoming material is in a wide range of formats, *Word* predominating, but all requiring extensive editorial conversion and preparation.

He uses a mixture of tools, including wordprocessors, formatters, typesetters, and layout programs, but he still does most final production in an ancient version of *PageMaker* because he's most familiar with it, although he knows that he will have to change to something else sooner or later.

This is someone who edits but rarely writes, and has to deal with a very wide range of quality of incoming material. Multiple outputs are a standard requirement, so the formatting or markup has to be adaptable to the purpose. Skill with multiple interfaces isn't a problem, but he wants them to 'make sense'.

Carrie: is a university administrator handling study-abroad programmes and their students. She has extensive correspondence with foreign institutions as well as reports on performance, conduct, and finance, parts of which are subject to discovery under a Freedom of Information Act as public records. Her background includes an MA in English Literature, so she writes fluently but has had little or no exposure to such aids as stylesheets or document structure.

Most of her work is done using a wordprocessor-like front-end to the old and cumbersome university admin document system, or with Microsoft *Word*, both of which she finds tedious but unavoidable. She is uninterested in what goes on behind the interface so long as her output is accessible in the right place at the right time.

She's aware that preparing documents is taking far too long with old software. A lot of material is duplicated, or nearly so, from document to document. Accented characters sometimes cause problems. Freedom of Information (FOI) requests mean accurate subject and title indexing is essential.

David: is a successful science-fiction author with a background in journalism, theatre, and IT. With half a dozen novels and numerous short stories published, he has a constant stream of articles to write as well as his current book in progress (and another two in development) and two unfinished plays on the back burner. He has written his own filing system to keep track of the myriad characters, places, objects, dates, and events across his storylines.

As a former IT worker, he is well aware of the technology, and has used a number of wordprocessors, outliners, and 'assistants', but switched a few years ago to Apple's *Pages* as his main writing tool, although his publisher requires *Word* for change-tracking. He knows that change is the only constant, and is keeping an eye on other systems, both from a writer's point of view as well as for technical interest.

He's impatient with technology that puts barriers in his way. As a writer, he sees structural constraints as barriers, not enablers, but understands why others need them. He knows that conversion to other formats requires a strong structure to be successful.

The notes from the discussions referred to in Figure 1.14 on page 50
are summarised after each persona in italics.

Figure 4.1: Revised personas

4.3 Semantics of the selected functions: What we must test and how

In the list beginning on page 212 we listed 14 topics that were derived from the requirements identified during the surveys of experts and users and the analyses of software and user queries. These topics relate to functions or aspects of the editing interface which we now examine in more detail to see if there are changes possible which might improve the acceptability of the interface among the target population.

Three of these topics (WYSIWYG display, Unicode conformance, and the maintenance of validity) have already been identified as being *a priori* requirements. They are in any case conceptual, rather than aspects of an interface that are subject to manipulation in the way that starting a new chapter would be. There are certainly different ways in which an alternative (for example, non-WYSIWYG) display might be turned on or off; different ways in which Unicode conformance might be temporarily disabled by an expert for problem-solving; and different ways in which the same could be done for validity-checking for the same reason; but these are largely binary choices (you either use it or you don't), and of little interest to the conventional user, so we shall not be investigating them further here.

Among the remaining topics, we combined some which are closely related, and we split others whose components were more effectively dealt with under another head. This resulted in the list below to be examined for testing. A few sub-topics were excluded, either because of their specialist nature (for example, Mathematics) or because they are essentially 'hygiene' factors: while they contribute to dissatisfaction for the experts requiring them, they are more connected with implementation factors than with the usability of the interface itself. Some discussion of them may be found in the notes on Implementation (section 6.3 on page 341).

Keyboard shortcuts (non-menu items): Four keys were identified in 'Keyboard Shortcuts' in the list above (page 215) for which there are already expected modal behaviours. Most non-alphanumeric keys behave differently in different circumstances, or represent different features, which can have a major effect on the markup as well as the user's experience and expectations. They also behave differently in different editors and on

different platforms, as shown in Table 3.11 on page 150 and discussed in the following subsections:

The Enter (Return) key [p. 228]
The TAB key [p. 234]
Backspace and Delete [p. 240]
The Spacebar [p. 238]

These characters do not have separate tests: instead, they are monitored during the other tests, and special dialogs are used to handle their usage. The markup characters of XML (\& and <) and LaTeX (\) are also considered here [p. 242].

Creating and opening documents: The File Open function is conventionally split into New Document [p. 244] and Open Document [p. 245], but only the first is tested. Opening an existing document is *either* completely non-problematic (it 'just works'), *or* it involves a significant level of expert intervention to set up the DTD/Schema and stylesheet, which would place it well outside the scope of this inquiry. Both, however, may incorporate some aspects of the Metadata functions [p. 245].

Test:section 4.5.4.1 on page 287, 'Start a new document'

Insertion: This includes both the External Files function [p. 250] and the addition of structural elements in element content: the Insert, Sections, and Structural markup functions [p. 246].

Test:section 4.5.4.2 on page 287, 'Insert a fragment from another document'

Test:section 4.5.4.3 on page 287, 'Add new paragraph after the current one'

Test:section 4.5.4.7 on page 288, 'Add a new section to the document'

Test:section 4.5.4.11 on page 289, 'Add a new numbered list to the document'

Styling (formatting) controls: All the Styling functions, with particular emphasis on typographic changes [p. 253]. The effects of this on the use of stylesheets is discussed in section 6.3 on page 341.

Test:section 4.5.4.10 on page 289, 'Make a word or phrase italic or bold'

Block moves: This includes some of the material from the Editing, Structured Editing and Navigation functions [p. 260]. It does not include a test for

hierarchical change, as this is dealt with in the section on SI.

Test:section 4.5.4.12 on page 289, ‘Mark and move an integral block of text to another location’

Links and referencing: Footnotes/endnotes, cross-references, and bibliographic citation and reference [p. 269].

Test:section 4.5.4.13 on page 290, ‘Add a cross-reference to somewhere else in the document’

Test:section 4.5.4.14 on page 290, ‘Add a citation’

Editorial: The remaining editorial functions (reorganising [except block moves], searching, spell-checking, and the adjustment of the toolbar functions) [p. 279].

Test:section 4.5.4.4 on page 287, ‘Split the current paragraph’

Test:section 4.5.4.5 on page 288, ‘Join the current paragraph to the preceding one’

Test:section 4.5.4.6 on page 288, ‘Join the current paragraph to the following one’

From the discussions below we synthesise in section 4.4 on page 281 the changes to the prevailing interface paradigms that we will test. There are several related technical topics that impinge on this, mostly concerned with how the proposed changes could be implemented (section 6.3 on page 341). These are not interface items to be tested, but principles or algorithms that would be used in implementations, and include:

Target Markup Adoption (TMA) [p. 342]
Smart Insertion (SI) [p. 344]
Preparation (adding new DTDs/Schemas) [p. 356]
Unstructured editing [p. 353]

In order to develop suitable material for testing, it was necessary to evaluate in more detail the current usage of each function, to try and ensure that we were representing the most obvious possible path to successful completion. In order to minimise the amount of guesswork, the personas which were re-evaluated earlier (section 4.2.1 on page 220) were applied to the task of deriving the scenarios for use in testing.

The basic scenario itself is straightforward: the task of writing or editing a document which is constructed of sections and subsections of paragraphs and lists in the conventional manner of a structured document (see the example in Figure 4.2). It could equally have tables, figures, quotations, or other elements of the Pool, but as we are not testing functions related to these, none are exemplified in the scenario.

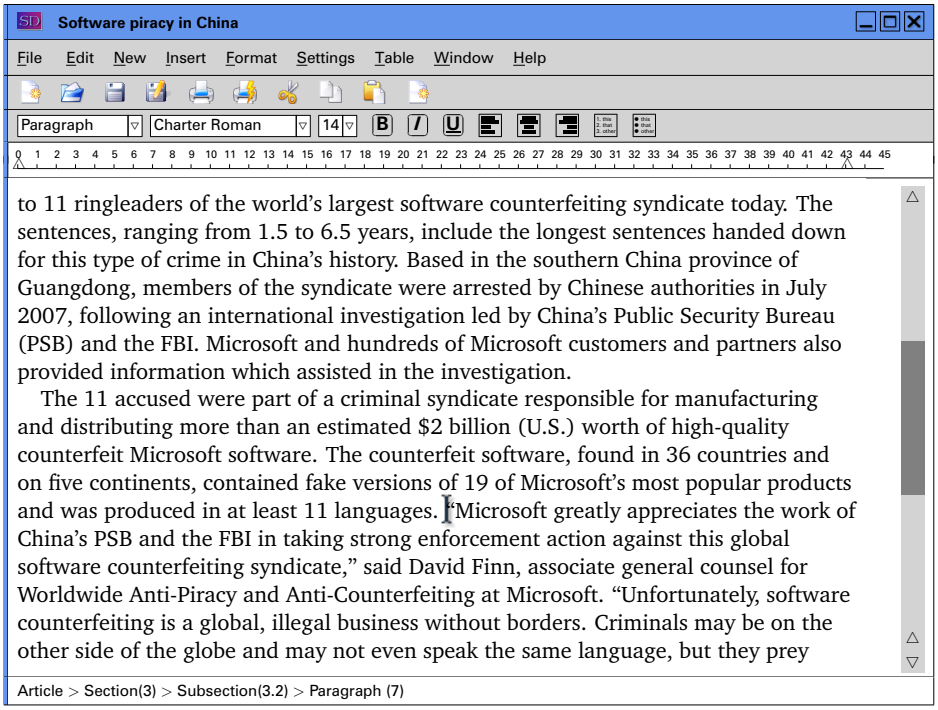


Figure 4.2: Example of the base scenario screen

A number of these texts were constructed to provide plausible contexts for the test sequences. The actual content and meaning of the text itself is not relevant, provided the material surrounding the actual cursor position which is the subject of each test is internally consistent. The final choice was an article on software piracy which exhibited the required attributes (Microsoft Corporation, 2008).

Adhering to the layout principles outlined in section 4.2 on page 217, we wished to make as few changes to the expected behaviour as possible, but to use what leeway exists to detect or infer user intent or to deflect unwanted or damaging actions. Detection of intent is particularly problematic when performed at third hand (that is, by the computer alone, without asking the user or the designer for help). Reliance must be placed on heuristic models (derived from measurement and analysis of past behaviour), and on cognitive deduction based on the context. This is simpler in low-level applications where the user is expected to have learned and become intimately familiar with each instruction, so that they have

no need for reliance on the program performing its own higher-level guesswork. However, as we have explained, experts in XML and LaTeX are not in our target population for an 'easy-to-use' structural editor.

The goal is to allow the users to continue with their learned behaviour wherever possible, and place the onus on the program to 'do it right',[4] whatever 'right' may be in the context of the action. This in itself involves an act of judgment, and in cases where the choice is ambiguous, an alternative strategy is suggested.

The following subsections analyse in detail the applicability, expected behaviour, and amenability to modification of the selected functions. The building of the model is described in section 4.4 on page 281 and the testing in section 4.5 on page 283.

4.3.1 Non-menu controls (keyboard shortcuts)

In other aspects of the interface, a designer or implementer can add, rename, and remove menu items, toolbar buttons, and other control surfaces. On the keyboard this is not possible, so an alternative approach for handling problem keys must be adopted.[5]

A western-culture computer keyboard conventionally contains the following groups of keys:

1. letters (26 in the English-language layout, more in others);
2. digits (10);
3. punctuation symbols (usually 12 keys);
4. function keys (10 or 12; 20 or more on some Macs and mainframe keyboards);
5. system controls (Escape, PrintScreen, ScrollLock, Pause/Break, NumLock);
6. modifiers (Shift, Control, Alt, Cmd [Mac], AltGr, and Fn [laptops], Compose [Unix] and Meta [Unix]);
7. white-space keys (Enter, Space, TAB, LF [Unix]);

[4] This was a recurrent phrase in the interviews and surveys in Chapter 3.

[5] In fact there are several reprogrammable keyboards on the market using Light-Emitting Displays (LEDs) embedded in each key surface to allow them to show the character or function they have been assigned. They are, however, prohibitively expensive.

8. editing or navigation controls (left, right, up, down, PgUp, PgDn, Home, End, Delete, Insert, Backspace);
9. a numeric keypad, duplicating the digits and some of the punctuation for convenience of data entry operations, and including an Enter key (in most systems functionally identical to the Return key).

A proprietary and largely redundant 'Windows' key exists on many keyboards, and sometimes a programmable Menu key. On keyboards targeted at the domestic or novice market, special keys for email and browser controls may also exist.

The printable-character keys (letters, numbers, and punctuation) are conventional and unproblematic, as are the system controls, modifiers, the numeric keypad, and the directional keys (the edit controls minus the Delete and Backspace keys). While the meanings of some of the lesser-used of these keys may be undetermined (and some like ScrollLock, Pause/Break, and NumLock are nowadays almost entirely redundant), the majority have absolute and invariant meanings, even within the modal contexts, and cannot sensibly be interfered with. The function keys are now largely unused except in specialist programs or by a user's own reprogramming, with the exception of F1 (help), F3 (find), and F11 (full-screen), and even those are not adhered to in some applications. Some common control variants persist, however, such as Ctrl-I for italics, Ctrl-B for bold, and Ctrl-U for underline.

This leaves the white-space insertion keys Enter, Space, and TAB, and their inverse counterparts for removal, Delete and Backspace. As was seen in the Expert Survey and the User Survey, these are precisely the 'problem' keys which correspond to the most variable interpretations, both in user experience and user expectation, and in the mental models of the designers of editing software.

In examining the ways in which the software may be more closely aligned to user expectation, we therefore have very little room for manoeuvre on the keyboard. It is of course possible to assign new and valuable functions to hitherto unused *combinations* of keys, or even to usurp existing functions, but this means that specialist training will be required for users to learn the new keystroke combinations, and perhaps unlearn some combinations still used in other applications: it is a popular conceit of vendors that their own program will always be the user's sovereign application. There is also a risk of conflict with control combinations that may already be used at a lower level than the application, for example the task-switching or screen-switching controls assigned by default to

advanced window-management packages like *Compiz*.

Any attempt to modify context-sensitive behaviour brings us into conflict with the expectations inherent in various operational modes, tell-tales, notifications, breadcrumbs, modal *vs* modeless dialogs, and other means of detecting or signalling user intent or program response. As we said at the start of this section, other aspects of the interface can be changed, but not the keyboard, so detection and interpretation of what the user means must of necessity be probabilistic or heuristic, and based on the contexts in which the key is pressed.

We therefore discuss these four keys (taking Delete and Backspace together as one) in the light of the context and the user's awareness of it (which may be marginal or entirely absent).

4.3.1.1 The Enter or Return key

In command-line interfaces (and further back into the days of the punched card), this key originally signified the end of a line and thus the completion of an instruction or a data record. The older name Carriage-Return (CR) is taken from the equivalent function on a typewriter of returning the typing position to the left-hand margin, combined with a Line-Feed (LF) to advance the paper by one line — the mechanics of this action entered the telegraph and Telex, and persisted from there into the early computer printer-terminals (Teletypes). Both CR and LF survive in the ASCII, the fundamental computer character set, but Unix-based systems use only an LF to terminate a line (where it is referred to as a **newline**), whereas Apple Macintosh systems used only CR until the adoption of the Unix-based OS X. MS-DOS and Windows systems continue to use both (CRLF). In XML, either character or combination is converted to a single LF during the multiple-white-space elision process known as **normalisation**.

Performing the New line function is still the key's primary task in console interfaces, but elsewhere the perceptions of its utility and its actual function are now deeply divided. In graphical interfaces (particularly web browsers) where the 'Task completed' operation is performed by clicking on a graphical OK, Next, or Submit button, it is still possible to use the Enter key to do the same[6], but in text-editing interfaces there has been a much more significant change, one that is

[6]Only in more recent interfaces such as those generically termed 'Web 2.0' is the function slowly being usurped by other means, such as the detection of the pointer being moved away from the focus, or the selection of another function.

directly related to perception and markup. It will be useful briefly to trace the development of this, as it parallels much of the development of the other characters we will examine.

1. In early editing applications, there was conventionally a line (record) length of 80 characters, derived from the punched card but also limited by the width of the printing-terminal page and the video monitor, both of which owe their original dimensions to the width of office paper. There was no concept of **wrap-around**: as with a typewriter, a bell or beep would sound when you got close to the end of the line, to signal that you might have to make a decision as to whether or not another word would fit, and if not, press the Enter key to go to the next line.Line-wrapping and related text-rearrangement functions were developed later to handle overruns automatically by inserting a newline at the last word-space before the line-limit, and moving the remainder of the line to a new line by itself. Software for text-handling tended to be for editing existing data rather than for writing new material: bulk text was prepared elsewhere, often offline, using skilled data-entry typists.

2. In the transitional period from plaintext editor to combination editor/wordprocessor, the Enter key risked becoming ambiguous: did it terminate an editing command, or was it just a means of getting to the next line? (see item 5 on the next page);

    - One solution was dual-mode editing, where one special key or combination had to be pressed before entering text-input mode, and another one to leave it (an early example was *TECO*, and the concept survives in the *vi* editor, still shipped with every Unix-based system).
    - The other was the invention of the modeless ('synchronous' or 'direct-intervention') editor, in which the alphanumeric and punctuation keys insert their letter or symbol straight into the file by default, and special or 'control' keys such as Escape and Control were reserved for editing functions — a feature which is still with us. The canonical example of this mode is the *Emacs* editor (Stallman, 1981).

3. With faster processor speeds, more memory, and better display drivers came the introduction of wordprocessors capable of real-time line-wrapping, so that a whole paragraph could be stored internally as one long line of (effectively) unlimited length, and only formatted into *apparently* separate 'lines' when displayed on the screen — it was still stored in memory as a

single long line. The advantage was that this dynamic reformatting would work in any selected width as the document margins or window geometry were changed; and textual additions, deletions, and other edits would cause the paragraph to be reformatted dynamically.

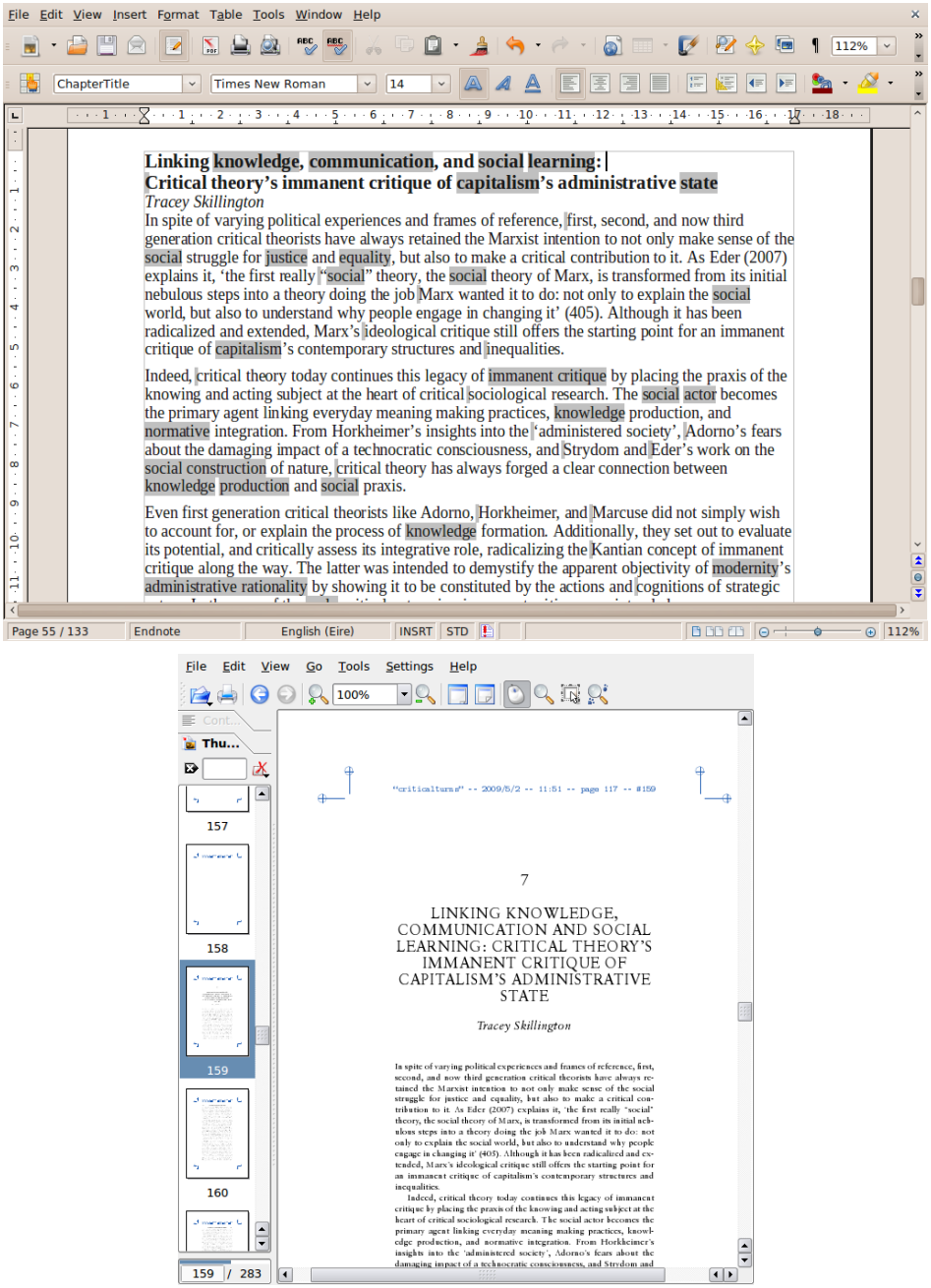
4. With the addition of controls for fonts and formatting in wordprocessors using hidden control characters, the storage format of documents was by this stage usually a complex binary file format, not physical ASCII 'lines', and because the CR/LF characters were no longer needed as line-break delimiters within a paragraph or at its end, the Enter key was no longer needed to insert them, but it could instead be used as the signal that the *paragraph* was ended, not the line.[7]

5. Pressing the Enter key thus came to mean 'start a new paragraph', but the user's perception was (and in many cases, remains) one of 'start a new line', as we saw in item 2 on the previous page. This conflation hides the fact that the two are clearly not the same, but the default formatting in some wordprocessors elides the visual difference between line and paragraph by removing *both* the first-line indentation on a paragraph *and* the space between paragraphs. This typographic solecism is avoided by professional document designers precisely because it makes the text difficult to read, especially when the paragraph is not justified, but this long escaped the default layout designs of wordprocessors. The distinction between line and paragraph is therefore seriously indistinct in the end-user's mental model because the difference between line and paragraph may be invisible — pressing the Enter key achieves the goal of starting a new line, but it has in fact started a new paragraph.

This last problem is clearly demonstrated in a frequent occurrence encountered by [human] editors: an author may introduce what looks like a line-break in long span of text (for æsthetic layout reasons) by pressing the Enter key, but has in fact split the element into two, creating (for example) *two* titles (Figure 4.3 on the facing page).

As it became impossible in the default layout to tell where one paragraph ended and the next one started, users now had (and, from observation, continue to have) at least three reasons in their mental model to press the Enter key:

[7] In plain-text editors, with no hidden markup, they key retained its meaning of 'start a new line' and the character retained its function as an end-of-line marker.



From 'Critical Turns in Critical Theory', Séamus Ó Tuama (Ed), Tauris Academic Studies, 2010, ISBN 9781845115593. Used by permission of the editor.

In the wordprocessor (top), the chapter title was broken after the colon (with the Enter key) to make it fit more neatly *in the wordprocessor window*, despite the fact that the typeset version would use wholly different margins and typefaces. This resulted in *two* titles because the Enter key invisibly broke the title into two paragraphs, each with the 'Chapter Title' style.

In the typeset version (bottom) the publisher's style did not allow for subtitles, and the narrower width and larger type meant the break after the colon could not be used, so the careful but unnecessary styling applied in the wordprocessor had to be undone manually.

Figure 4.3: Effect of using Enter to break a line prematurely

1. to start a new paragraph;
2. to add extra space between paragraph-level objects with second and subsequent presses of the key to insert empty paragraphs, known as 'blank lines' (in reality, empty paragraphs);
3. to break a 'line' prematurely for reasons of manual formatting, readability, or æsthetics (neither knowing nor caring whether this is a new line or a newly-created paragraph).

While the second reason is understandable in the absence of a properly-designed stylesheet, the third is clearly an example of desperation by the user in the absence of any other means of expressing intent, which has a very close bearing on this research: when margins are later adjusted, or text added or deleted so that the text is reformatted, any premature breaks within the paragraph remain, sometimes in mid-sentence. This destroys the formatting, and all such manual line-breaks need to be removed and the text joined back up again, incurring a heavy penalty to the author, publisher, or typesetter in lost productivity (as in Figure 4.3 on the previous page).

As this behaviour, both by programs and users, is still — from observation — commonplace, we need to detect how the key was used for each task; we approach this from the user's perspective — previous analyses tend to approach it from the point of view of the technology (Quint & Vatton, 2004, p. 118).

In the rare cases where a 'real'[8] premature line-break is required within a paragraph, without terminating it, most systems have the facility to insert one (*Word*'s Insert|Break|LineBreak, HTML's `<br/>` element, or LaTeX's `\\` command), but while the HTML and LaTeX versions are commonplace, and known to every user, the equivalent in wordprocessors seems rarely to be taught in training courses, and even more rarely used. Separate paragraphs are used instead, again making subsequent processing or reuse impossible without extensive post-editing.

It must be noted that XML editors would only be able to implement this robustly if the DTD or Schema provides markup for the purpose (such as HTML's `<br/>` or TEI's `<lb/>`). In this author's experience, many XML vocabularies do not provide the markup in element form, meaning that a Processing Instruction or Character Entity is required to mark the location for later processing. The reason conventionally given is that 'XML markup is not for formatting', and that HTML

[8] A suggested measure of 'real' in this circumstance would be a line-break that would survive a reformatting operation and still be in the correct place. An example would be the separate lines of an address laid out vertically within a single paragraph.

and TEI are exceptions because of the needs of their particular constituencies — HTML serves people with no background in formal markup; and TEI provides facilities for transcriptions of literary and historical documents, where preservation of the original formatting may be of primary importance.

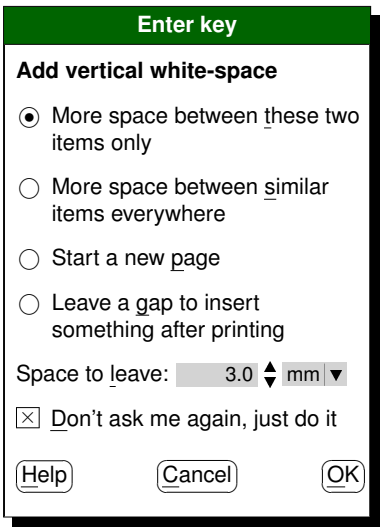
A fourth reason for using the Enter key also exists: multiple consecutive presses to create a blank space; and this is very much tied to the detection of intent. A number of SGML and XML editors (for example, the long superseded STiLO *Document Generator* and *WebWriter* editors) used the first press of the Enter key in the conventional manner as an instruction to start a new instance of the existing element type (for example, a new paragraph, list item, etc); however, an immediate second press (with no intervening key) replaced the new element with the next choice in the current content model; third or subsequent consecutive presses cycled through the remaining available element types in the current content model. A skilled user could therefore use this to terminate the current structure (for example, a list), and rapidly select the next required element type simply by using the Enter key the correct number of times (the currently-selected element type was shown in a tell-tale).

A similar method is used in *Emacs/psgml* with `C-c RET` (Ctrl-C Enter) to start a new instance of the current element type on the first press, but on successive presses it splits the next ancestor element if possible, in order to ascend the document hierarchy rather than to select cyclically from the current content model.

In *Word*, the use of Enter at the end of a list item follows this pattern by creating a new item, and a second press will negate the prevailing list item style and revert to a plain paragraph by removing the indentation and the bullet or number. However, at this stage it can go no further, because *Word* has a flat document model, so subsequent presses merely create additional empty paragraphs ('blank lines'). This appears to be one of only two circumstances in a wordprocessor where an unmodified keypress has a significant effect on the document markup.[9] Pressing Enter at the start of a paragraph has the same effect: an empty paragraph appears above the current one. However, in the example of a list item, as Backspace can also be used to remove the unwanted list item label (bullet or number) and its associated indentation from a newly-created item, it is unclear

[9]The other one is the facility in a *Word* style to set the default style for the next paragraph; so that, for example, terminating a Title style by pressing the Enter key will set the new paragraph the Author style.

how extensively either method is taught or used.



This dialog would appear when the user presses the Enter key for the second successive time (assuming that the interface does not already have an action assigned to this) or presses Enter at the start of a paragraph. The objective is to identify what formatting the user wants, and to allow that to become the default.

Figure 4.4: Example modal dialog for adding vertical white-space

For the addition of vertical white-space, we need to monitor and detect multiple consecutive presses of the Enter key, and single or multiple presses when the cursor is at the start of a paragraph. The test suite includes a modal dialog for use in these circumstances, where the user can select the reason, as in Figure 4.4. To minimise excise on the established convention, when this is triggered by the second of two Enter keys in succession, a third press would select the pre-selected 'here only' option [1]. An implementation could choose to insert, for example, twice the current default inter-paragraph spacing or a single line-height's worth of white-space, thereby giving the expected default at minimal excise (one extra keypress).

4.3.1.2 The TAB key

It is important to distinguish the TAB *key* on the keyboard from the TAB *character* (ASCII decimal 9) which may be present in a document: while they are connected, they are not necessarily the same thing.

The TAB *character* is an anomaly because it has no fixed visual representation — printable characters have a glyph; a single space is a single space; and even most non-printing control characters have a fixed, if irrelevant, visual representation, even if it is just an empty rectangle. Instead, the existence of a TAB character in a

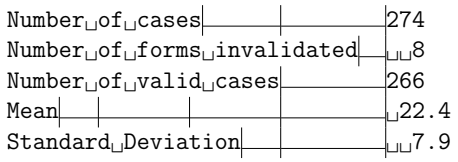
plaintext file usually causes the character following it to be shifted rightwards *when displayed*, to the next position which is a multiple of some number of spaces (traditionally seven) from the left-hand margin, or to a location pre-set by a tab-stop or margin delimiter. All remaining text on the line moves rightwards after it (the polarity would be reversed for right-to-left writing systems). However, only a single character is stored.

While this can be used in plaintext to make blocks of text line up visually when the file is displayed, the same effect could also be achieved by judicious use of multiple spaces, whereas the spacing as we describe it is actually the effect of a single character being interpreted. In some editing modes the character is actually converted on entry to the requisite number of spaces anyway, to avoid storing files containing actual TAB characters.

While such interpretive behaviour is perhaps useful for the skilled plaintext user, who achieves the alignment expected, it is clearly risky for the programmer or data manager, for whom the TAB character may be an important delimiter or data value. While this interpretive behaviour contributes to saving subsequent editing time (multiple spaces take more keypresses to delete than a single TAB character), it also happens at the expense of subsequent processes which may rely on the TAB character to *symbolise* the need for alignment, rather than actually *perform* it. Despite appearances, such a TAB character is not usually intended to perform tabbing to a location: to do that accurately with variable-length field data would require *multiple* TABs between some values, as shown in Figure 4.5 on the following page. This can be viewed as an example of the distinction between descriptive and procedural markup.

This problem is evident in a manually-edited file, not intended as TAB-separated data, when the value preceding the TAB character may be a field of very variable length on different lines. The user will enter *multiple*TAB characters (and perhaps spaces as well) to follow the shorter texts, in order that the remainders of those lines still align with the tabbed material on other lines with longer texts (see Figure 4.5 on the next page).

Some wordprocessors employ a similar system, using a tabbing mechanism based on that of a typewriter, where the tab-stops mentioned above can be set at predetermined locations rather than a built-in multiple of characters. However, in this case, a single press of the TAB*key* will cause the cursor position to move to the next tab-stop position in a single move, regardless of how far away across the page the next tab-stop is located. Because the GUI hides the underlying action, it



Each TAB character entered causes the cursor to move to the next position which is a multiple of seven spaces from the left margin. The remaining alignment of a numeric 'column' is done with individual SPACE characters (shown here as small open-topped squares).

Figure 4.5: Manual alignment using multiple TAB characters and spaces

is unknowable to the user in any given program what is actually being *stored* (should they indeed care): a single TAB character? (with reliance being placed on the continued persistence of that tab-stop position); multiple TABs? (and perhaps spaces); or — the most useful — a more complex internal instruction (markup) to compute or signal a move to the next defined tab stop or column boundary?

In informal markup systems such as wordprocessors, a difficulty typically arises when a document created by an unstructured system is opened or imported into a structured one, requiring interpretation and conversion of TAB *characters*, where different font spacing and margin settings may cause the entire alignment to be interpreted differently.

Formal markup systems do not have this problem, because columnar alignment requiring a tabular setting is precisely specified by element types in XML such as `row` (`tr`), or `entry` (`td`); or where a special column-separator character is robustly bound to a tabular preamble which describes the column-formatting, as in the case of LaTeX. In these systems, pressing the TAB*key* may not even have an assigned behaviour (we consider below the possibility of deactivating or usurping this key).

This location-dependent formatting, largely inherited from the typewriter, means that the TAB *key* carries a particular connotation in the user's mental model of 'go to the next "column"', even when there is no formal definition of columns in effect (for example, at the start of a paragraph, in order to achieve indentation). The result is that multiple presses of the TAB *key* are widely used to attempt alignment which should properly be done with a Tables mode or by using a margin control. Because the GUI appears to have done what the user expected, the unreliability of the result is only seen when post-processing is attempted, or

when the document is opened in a different environment, as a result of the variable interpretation described above and illustrated in Figure 4.5 on the preceding page. As with the [ab]use of the Enter key, this can result in very significant time and cost penalties in post-editing.

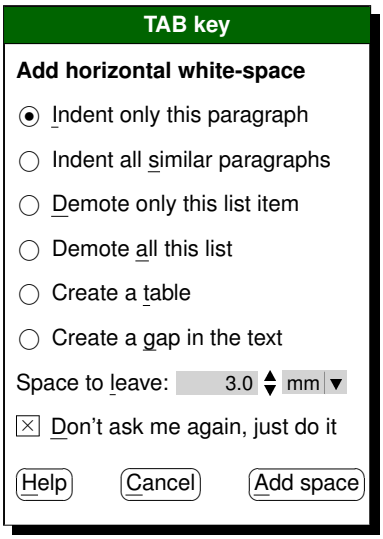
In spreadsheets and web forms, the TAB *key* is used to move the cursor to the next cell or input field, and no character is inserted or transmitted.[10] This serves to confirm the user's mental model that the TAB *key* can be used successfully to move to the next 'column'. Another common use for the TAB and Shift-TAB keys is to perform demotion (indentation) and promotion (undentation) respectively of lists and other indented or indentable material.

The TAB *key* is therefore viewed with hostility by programmers and document engineers because its use to insert multiple (and uninterpretable) TAB *characters* into the document can destroy the integrity of the document. When the display of tabular alignment in a document is governed by a stylesheet acting on markup, there is no need for the TAB at all (neither the key nor the character); and in a table-editing or form-editing mode it can be used as described above to skip to the next cell or field. Its use for indentation is redundant, as indentation is controlled by the relevant markup and styles for lists, paragraphs, block quotations, and other elements requiring distancing from the left margin. This leaves the key free to be detected at the start of a paragraph and elsewhere, and used to invoke a modal dialog to change the current location or indentation style, as shown in Figure 4.6 on the next page.

The key could, however, be used in non-tabular environments to invoke the Create table function, or to perform block indentation as already mentioned above, on the basis that its use is most likely to be associated with the need for an alignment of some kind, but we do not explore this further.

This leaves open the question of authorial demand for *ad hoc* spacing, such as an intentional chasm (the editorial representation of a gap in a document being quoted), or the simple setting-apart of two objects for visual symmetry (images in a figure, joint author signatures, etc, which is really an alignment problem). Such occasions should be provided for in the available markup, and are not examined here.

[10] Physically storing a TAB character in a web form field can sometimes be achieved by copying and pasting, resulting in corrupted data in Common Gateway Interface (CGI) Web scripts which use the TAB character as a TAB-Separated Values (TSV) file format column separator!



This dialog would appear when the user presses the TAB key outside a tabular or forms-design context. The objective is to identify what formatting the user wants, and to allow that to become the default.

Figure 4.6: Modal dialog for adding horizontal white-space with the TAB key

4.3.1.3 Spacebar

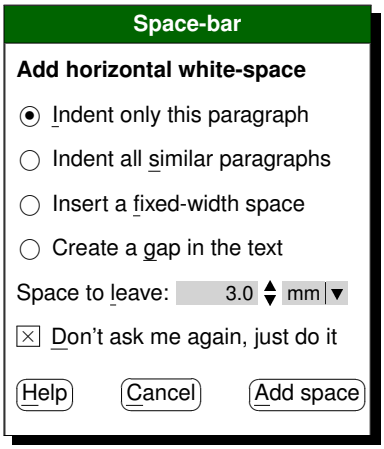
The primary use of the Space key (spacebar) is to insert a single space character, but it is also used in Unix-based console (terminal) interfaces to implement a Continue function when paging through a document; and a related Select function in widgets. The paging function was used by default in Tim Berners-Lee's original console-mode web browser *www*, and it persists in modern browsers as a Page Down function.

The normal space is the most common of the white-space characters (TAB, space, newline, form-feed, and the joining, non-joining, breaking, and non-breaking characters and spaces [several widths] of the Unicode character repertoire). In most XML and LaTeX document applications, multiple consecutive white-space characters in mixed content can (configurably) end up being normalised to a single space (white-space in element content is irrelevant in XML when a DTD or Schema is being used). The major exception is the TeX engine's interpretation of two or more consecutive newlines as a paragraph break. A normalised space may then be interpreted on output (for example, typesetting) as the flexible quantity needed to achieve spacing or justification, rather than as the typewriter-style fixed-width character of a tenth or twelfth of an inch. By this point, the concept of '*a* space' has relinquished its meaning, and become simply 'space'. This behaviour is the default in HTML in the browser; in LaTeX during typesetting; in XML applications using the `normalize-space()` XPath function; in XSLT when the

`strip-space` setting is in effect; in XML editors using similar settings, and in XML documents where there are specific attributes such as `xml:space` to control the interpretation of white-space characters.

However, as we have seen with the newline for vertical white-space and TAB for horizontal, users have become conditioned by wordprocessor interfaces to assume that creating horizontal white-space by repeated insertions of single space or TAB characters is the *only* approach, because it 'works' in wordprocessors.[11]

An implementation would therefore need to monitor and detect the use of multiple spaces and enable their replacement by some other form of placeholder, in a similar manner to the detection of multiple presses of the Enter key already mentioned. The implementation of space-insertion is problematic (see the discussion in section 6.3.6 on page 350), but the detection of intent is straightforward: the objective must remain to make it as unobtrusive as possible.



This dialog would appear when the user presses the Spacebar for the second successive time (assuming that the interface does not already have an action assigned to this). The objective is to identify what formatting the user wants, and to allow that to become the default.

Figure 4.7: Example modal dialog for adding horizontal white-space

A special case is the typist's use of two spaces between sentences. In the days of typewriters and early wordprocessors, this was necessary because in monospace typefaces, a single space provides insufficient optical separation at the sentence boundary. TEX detects a lowercase character followed by sentence-ending punctuation and white-space, and automatically adjusts the inter-sentence gap to be slightly larger than the normal inter-word gap calculated to justify that

[11] Significant surprise awaited new users of the SGML/XML editor *Author/Editor* 2.0 (1990): second and subsequent presses of the space or Enter key in mixed content would cause a beep, and no cursor motion.

paragraph. While a similar form of detection could be used in editors, the effect is not normally noticeable by the untrained eye. It needs to be emphasised that any modal dialog should trigger on multiple spaces *only* when the preceding non-space character is *not* a sentence-ending punctuation mark, *and* when the character before that is not lowercase (that is, the logical complement of LaTeX's method). However, the presence of multiple spaces at a sentence end is normally not problematic, precisely because of their elision by subsequent processes. No such detection was therefore planned for inclusion in the tests.

4.3.1.4 Backspace and Delete

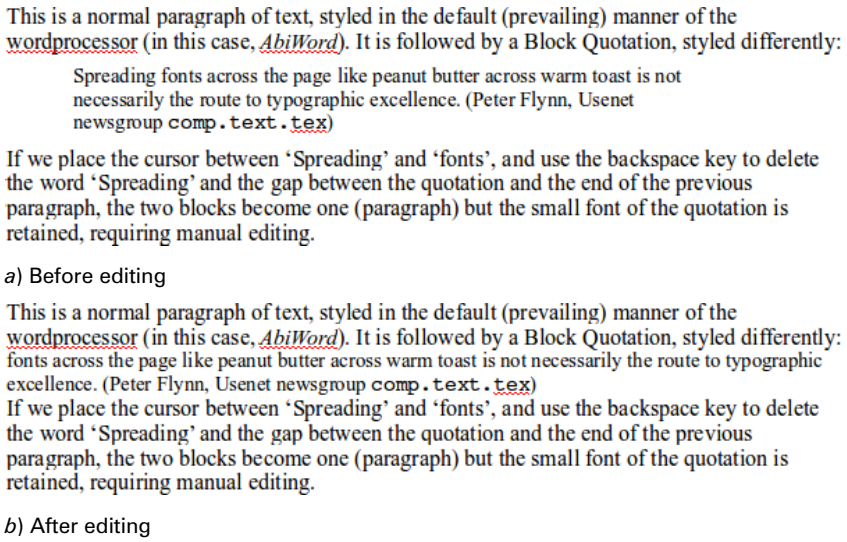
The Backspace key deletes character data leftwards; the Delete key does so rightwards (reversed for R–L writing systems). In some earlier console applications (on VAX/VMS and Unix systems in particular), the Backspace key used to behave like a left-arrow key, moving the *cursor* leftwards but not deleting any characters. In older Teletype applications, the interface did not allow the editing of text within a line being typed, so the destruction of previously-typed characters was not possible: in these cases the console would do nothing, or perhaps echo the symbolic notation for Control-H (`^H`), which is the keycode representing a Backspace character in the ASCII character set.[12] Quint and Vatton (2004) appear to conflate the Backspace key with the Enter key, taking the view that it is used to merge adjacent elements in the same way as Enter is used to split them. We describe this action below, but as a consequence of progressive deletion leftwards to the element boundary, not as the primary function. This appears to be supported by Owston et al. (1992, p. 264), who found that students used Backspace mainly and extensively for deletion of text (as well as for structure) instead of marking the text block and using a single keypress of Delete or Backspace. Notoriously, some Apple Macintosh keyboards have the normal Backspace position (above the Enter) key occupied by a key labelled, confusingly, Delete.

We do not consider here the use of Overwrite or Overtype mode, where characters

[12] It is worth recording that this behaviour has become mimicked in Usenet newsgroups, email, Internet Relay Chat (IRC) and similar channels, and elsewhere to represent the humorous mock-redaction of insults. This parodies naïve users who try to edit an undesirable word but fail to realise that what appear to be destructive backspaces on their screen are in fact merely inserted Ctrl-H characters. It also leaves evidence of the writer's actual feelings while pretending to provide plausible deniability for the presumed statement, for example: `He was being fu^H^Hextremely stupid.`

typed in the midst of existing text do *not* push the rest of the line rightwards, but instead replace the existing characters. This mode is rarely used today except in special circumstances such as the 'drawing' modes of *Emacs*.

In plaintext applications, all characters are data, so backspacing or deleting over line-ends, TABs, spaces, and other non-alphanumerics simply removes them. Similarly in wordprocessor applications, visible characters are removed until the boundary of a paragraph block (a newline or blank line) is encountered, at which point continued use of the key will remove the boundary between the blocks and join any text remaining on the non-destructive side of the cursor to whatever lies on the destructive side. In the process, the applicable styles may be adjusted, possibly unpredictably: when one type of paragraph block is joined to another in this way, and the styles differ, it is unclear which style should prevail (Figure 4.8).



In this example (using the *AbiWord* wordprocessor), the font size of a Block Quotation is retained when it is joined to the preceding paragraph: an example of an editing function which does not use SI techniques.

Figure 4.8: Retention of styles after merging paragraphs

In a structured environment, better control is possible because the markup can be used to decide what to delete. Each keypress must delete one character, with the cursor moving correctly to the next character in linear sequence, regardless of the depth of markup boundaries through which it has to pass to attain this location.

Only when an element has been emptied of text content by deletions, might the element node itself be silently removed along with any generated content, unless the element is compulsory in that location according to the DTD/Schema, in which case it must remain in the document structure and should acquire its default **ghost** (see footnote 3 on p. 147). When a EMPTY element, PI, or other

node which creates generated text is the subject of deletion in this way (and is not compulsory), that text must be treated as a single character, and both text and node must disappear from the screen and the document structure in a single keystroke. This preserves the integrity of the document but may conflict with the user's expected view, which is that pressing the Backspace or Delete key once must delete one character (or token, in the case of generated text on a single element type).

As we are considering only the case of a synchronous typographic editor, the case of deleting individual markup characters themselves (and thus rendering the document non-well-formed) does not occur. A good example of the problems encountered in implementing **handlers** (programming modules) for dealing with this level of editing — using *Xopus* — is in O'Connor, Gnanapiragasam, and Hepp (2013).

4.3.1.5 Markup characters

In XML, the only two characters with an initial special meaning are the less-than sign (`<`) which starts a tag or declaration, and the ampersand (`&`) which starts an entity reference. It is therefore critical in XML editing that these two characters above all others can never find their way into normal text in their raw form, but must be replaced internally with their symbolic notation as character entity references `<` and `&` when typed by the user or pasted raw from an external source.[13]

In WYSIWYG mode, trespass on the markup characters is not possible because they are not visible, so the opportunity for the casual or untrained user to edit the markup directly (and damage it) does not exist. In any case, in WYSIWYG mode, the markup itself tends not to exist in the canonical pointy-bracket or backslash form that is seen in plaintext views, because the document structure is actually held in memory by the program in an entirely different form (trees, maps, pointers, etc).

In those (usually plaintext) editors where access to the characters forming the

[13] Once a less-than sign or ampersand is read from an XML document, an entirely different set of meanings is assigned to several characters. For example, the exclamation mark becomes a declaration-open character and the greater-than sign becomes the tag-close character; or the semicolon becomes the reference-close character. But this only occurs internally, at the level of the lexical scan of a parser, and the user neither sees this nor has any influence over it.

markup is possible, it becomes the individual users' responsibility to exercise care in what they edit, which requires some expertise and dexterity — possibly the most compelling reason for not permitting the untrained user even to see that markup exists.

In LaTeX, there is a single character which introduces markup: the backslash (\). In a similar manner to XML, once this is encountered, an entirely different set of rules for subsequent characters comes into effect until the end of the command, but this is again an internal matter. It is only important for a WYSIWYG editor to ensure that any attempt by the user to enter a backslash as text causes it to be silently inserted symbolically as the **control sequence**`\textbackslash`.

However, TeX specifies nine other characters as 'special', in that they have typographic effects at odds with their ASCII definition when used in their raw form.[14] In plaintext editing they must be escaped with a backslash if the literal glyph is required; thus to obtain a $ sign the plaintext user must type \$. There are three additional characters[15] which cannot be used in normal text outside mathematics mode. Typographical WYSIWYG editing software must perform all the re-presentation of these characters, allowing users to type a dollar sign and see what they expected while silently inserting the \$ command.

Some editors may usurp other keyboard characters for special purposes: for example, *Emacs* in LaTeX mode responds to a first press of the unidirectional double-quote character (") by inserting LaTeX's code for typographic open-double-quotes (`` ` ` ``, which gets typeset as “); and to a second press by inserting the code for typographic close-double-quotes (`'' `, which gets typeset as ”). This procedure occurs automatically and cyclically for odd-numbered and even-numbered keypresses of the double-quotes character respectively much in the same way that some wordprocessors do.

There is no change to the interface envisaged for this requirement, as the rules are plain and the demands of the file formats are well-known.

[14] The dollar (math-mode toggle), percent (comment), caret (superscript), ampersand (column separator), underscore (subscript), tilde (non-breaking space), hash (macro argument reference), and the opening and closing curly braces (argument delimiters).

[15] The less-than, greater-than, and vertical bar.

4.3.2 Creating and opening documents

In any structured-document system there is a set of rules or conventions which can be used to guide the formation of the document. In the case of XML, the rules are formalised in a DTD or Schema, and the editing engine has to ensure that the rules are not broken by its own or a user's actions. In the case of LaTeX, the conventions are implemented as commands defined in the document class file, which can be used by the writer to mark the structural components (but commands from a different package, or those defined by the user, may be used instead). In LaTeX there is no concept of 'compulsory' or 'prohibited' in the sense of a DTD or Schema to guide the editor: any requirements for specific commands or environments to be present or absent can only be detected when the formatting engine is engaged, when an error message may issue if the requirements are not met.

4.3.2.1 New documents

Most editors examined (see section 3.2 on page 133) came with a selection of predefined or precompiled document types. These ranged from the four default LaTeX classes (book, article, report, and letter) to the dozen or more XML vocabularies provided with *Serna*, *EPIC*, *XMetaL*, etc. When a user requests a new document with the File|New menu or the New toolbar icon, a modal dialog appears, requiring the user to pick one of the known document types. Most systems also include a generic 'blank document' as a fallback, but as this provides no specific markup or styles, its use is limited. It may also be a contributory factor in user rejection of structured-document editors that the selection of prepared document types is inadequate for their needs.

It is clearly impossible for an editor to provide precompiled facilities for every possible type of document, or even for all known types. While some editors can be criticised for not providing a big enough range of document types (there is a strong emphasis on types designed for technical and scientific documents; only *oXygen* appears to provide the TEI in precompiled form, with stylesheets), the test is more properly whether or not the editor provides facilities for precompiling and installing new types of document. We exclude here the question of users in specially controlled or secure circumstances not being permitted this action.

No change is therefore required to the action of the File|New menu or New

toolbar icon, but two possible additions were considered:

- a New type of document function for those editors not already providing it (section 6.3.11 on page 355);
- a dialog for the specification of metadata (minimally title and author) as discussed in section 4.3.2.3.

However, testing both would require extensive dialogs, and procedures for undertaking the necessary preparation are already well-established in the software which handles them.

4.3.2.2 Opening existing documents

For document types which are not 'known' to the editor (that is, they do not exist among the precompiled document types mentioned in section 4.3.2.1 on the facing page), editors will usually read a DTD or Schema as specified in the document itself, or via a dialog with the user.

The exceptions to this are those editors which can only read either a DTD *or* a Schema, not both (most notoriously, Microsoft *Word* in XML mode cannot read DTDs); and those editors which are specially restricted from loading any document type apart from those on a carefully controlled list (this includes military systems, and editors used in critical documentation applications in industries such as as medical or industrial safety).

Where no predefined stylesheet exists for a DTD or Schema, a method is outlined in section 6.3.11 on page 355 which could generate a sufficient format for the user to start editing.

4.3.2.3 Metadata

The inclusion of metadata is an everyday event most usually hidden from the author in synchronous typographic interfaces. Image names, residing and positioning details, ID and IDREF links, and the identification of title, author, date, and other document properties are examples of 'information about information' that are commonly collected or stored in structured document files.

Explicit metadata, such as retrieval terms for cataloguing purposes, and the title, author, and date, are commonly added during the authoring or editing process,

although some items are required by the more rigorous document control systems before a new document can be created.

Unlike a wordprocessor, where a document's title and author may not be evident in the absence of descriptive markup, most XML and LaTeX documents identify author, title, date, and other metadata with the markup. There should therefore never be occasion for these items to be added manually in a Document Properties function, although for other metadata like keywords, subject, scope, and publication data such a facility must be made available.

No test was envisaged for the addition of metadata at the time of file creation, as this would just be a set of prompts or a form-fill pane. While such intrusions have their own set of usability criteria, this does not affect ongoing editing of the document.

4.3.3 Insertion of new material

We have shown in section 1.1.3.2 on page 20 that the use of terminology related to trees and 'insertion' is inappropriate in the context of non-markup-expert writers, and we discuss in section 6.3.2 on page 344 some ways of managing the writer's intent in requesting the creation of a new structural component when the rules applying to the current cursor position dictate that it is not possible at that point.

We therefore propose that both the avoidance of the terminology *and* the creation of new components in the appropriate location be covered by the use of a menu or toolbar item labelled New.

From observation among authors and editors, there is a virtually universal habit of referring to the action of adding another structural component to a document by the term **new**, as in 'new chapter', 'new paragraph', 'new item', etc. This usage is so common that it usually passes unnoticed, and it is also conventional among occasional writers, including students, clerical staff, and the casual or domestic keyboard user. Its origins remain conjectural, but may have been reinforced by the voice markup of the Dictaphone and stenographer era ('Full stop, new paragraph, Miss Jones').

As we mentioned in the section referred to earlier, the view of 'adding to a document' in the programming, systems design, computer science, and document engineering disciplines is quite different, and it was also recognised early on in

the Humanities Computing field, along with its limitations (Sperberg-McQueen & Huitfeldt, 1998). The model of the document is the tree, and the container structure inherent in the hierarchical model invites the conceptualisation of adding a component as one of **inserting** — the implication being that there is already some surrounding structure into which the new component is to be introduced.[16]

As a result, the addition of a new structural element in an XML document is almost universally referred to in editor menus as 'Insert Element' (see section 3.2 on page 133), a term which has no meaning to the author who does not have formal training in computer science or XML document structures. We therefore propose to abandon this practice in respect of structural elements, and label the menu New.

Perhaps perversely, however, we do retain the Insert function for two different categories:

1. the addition of elements in mixed content, where the use of the term 'insert' is standard even in common parlance (as in 'insert a comma after "sincerely"' or 'insert a footnote here');
2. the application of formatting adjustments like manual linebreaks and pagebreaks, hyphenation points, and other similar 'tweaks' (the term used in document engineering), but these are usually added *post hoc*.

As was mentioned in the fourth item on page 147, adding new elements in a WYSIWYG environment, especially compulsory elements in a new document, demands the provision of strong visual cues to their existence and location. In many editors, this is provided by **ghosts** of the element name or usage in the form of grey prompts which are overwritten the moment the user starts to type text in them (readers may have seen this technique in some web site search boxes, where the word 'Search' is displayed until a key is pressed).

The proposed use of 'New' as a menu comes at a small excise in formal terms. Johnson (2000, p. 206) explicitly warns against the use of verbs, giving two reasons: *a*) the word is not a verb, and command names, by current convention, *should* be verbs; and *b*) users of *most* software applications may not have

[16]The conceptual problem of inserting a document root element into the nothingness of an empty file was solved by reference to the **nodes** of the tree model in graph theory; in XPath terms this became an insertion into the abstract concept of a **root node**, which contains everything else (XPath, 1999)[§ 5.1].

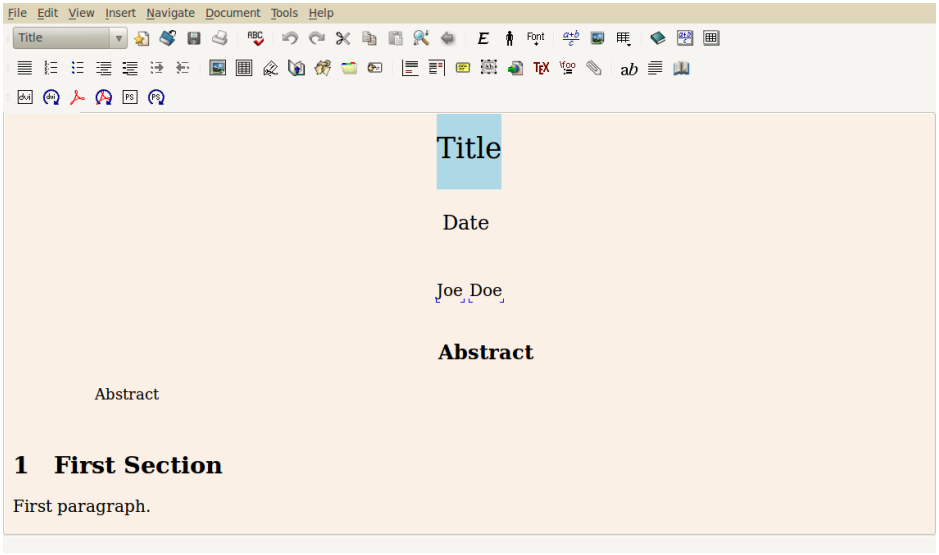


Figure 4.9: Ghosting of automatically-inserted elements in LyX

encountered the usage of a non-verb before — in both cases the italic emphasis is mine. This argument is possibly true in financial, technical, and some other business applications, but for the reasons given above, it is precisely *not* the case in normal document editing, or in general authorial practice, where 'New' has a distinct and well-established semantic. In a careful study of naming, Jørgensen, Barnard, Hammond, and Clark (1983) analysed and tested the complex field of different naming strategies, and found that designers were not consistent, but at least relatively systematic in their assignment of names (to be clear, this was a study of naming *commands*, in the days before widespread GUIs: rules for consistency were proposed in Green and Payne (1984)). However, these authors do cite Scapin (1982, p. 18), who distinguishes 'labelling' (the names assigned) from 'structuring' (the reuse of labels: according to the function, the object, and both), who found that although verbs were chosen only 20% of the time, verb-only labels led to less diversification through reuse. In our view the demands of semantics and obviousness are strong enough both to justify the use of New, and to override the advice on verbs.

A number of variants were considered, however, including 'Create New', 'Add', 'Add New', 'Insert New', etc, but they bring complexity or ambiguity to the menu title without achieving the directness of the single word.

The instantiation of this function in graphical terms for a toolbar icon is problematic, as it is conceptual rather than concrete. However, there is a useful analogy with the New File and New Folder functions, for which many patterns exist, and a + sign appeared the most appropriate.

There are two principal applications for this function, dealt with below:

1. the explicit manual addition of new elements, one by one, as writing or editing progresses;
2. the implicit addition, sometimes of several or many elements at once, when material is pasted from the clipboard or added from an external resource such as an included file or fragment.

4.3.3.1 New structural elements added one-by-one

Structural elements were defined in section 1.1.3.1 on page 16 by reference to the concepts of document hierarchy and pool: they cannot occur in mixed content.[17] In XML, the structure is formally declared in a DTD or Schema; in LaTeX (as in SGML) it can be implied by the nature of the markup, as explained in footnote 7 on p. 33. The reader is referred to the diagram of an example document structure in Figure 1.7 on page 19.

As we have described in section 3.4.3.4 on page 194, existing XML editors require the cursor to be in element content for the 'insertion' of a new structural element to be valid, although in some circumstances it can be deduced, as explained in section 4.3.1.1 on page 233 in respect of the use of the Enter key at the end of an existing element. In a validating editor, it is not possible to insert a new structural element anywhere other than where the DTD or Schema permits; if the cursor is elsewhere the option or the element type name is greyed out or simply absent from the menu, which is baffling to the non-expert user.

The use of a separate function, 'New', for the addition of new structural elements is a convenient way of implying that this interface restriction is irrelevant. The objective is that if the author requests a new chapter, section, table, figure, list, or other structural element, then that is precisely what should occur, regardless of the position of the cursor — not by inserting it in an invalid location, but by moving to the next valid location and inserting it there. This differs significantly from the stance taken by Quint and Vatton (2004), which involves cyclically splitting the current element until a valid insertion point is reached.

[17] There are a few exceptions to this rule: the TEI, for example, permits a new `text` element to occur in mixed content, for the purpose of allowing embedded whole documents, such as a letter or fragment being read out by a speaker as part of a speech (also common in transcriptions of parliamentary debates).

Given that a validating editor already knows the available markup at any given point in the document, it is possible for it to locate the next valid point for the new element to be placed, and simply move there and do it. This is the primary task of the SI technique described elsewhere. Currently only two XML editors (*Arbortext* and *Xopus*) are known to have experimented with this method.

There are two potential problems with this approach: the 'next available location' may not be the one the author wanted, or the author may have wanted to split the current element (and its ancestors) in order to create a new instance of the element *at that point*.

The first is a valid objection in the case of hierarchical elements: if the author requests a new section, it may not be wanted after the end of the current section, but at the end of the chapter, after all existing sections, if that is a different location; or conceivably, earlier in the chapter or even elsewhere in the document. However, it would appear from observation to be habitual to position the display and the cursor at the location in these circumstances.

The second is perhaps less common: its expression in words (such as an instruction to an editor) would be along the lines of 'split this section into two new sections', implying that the current one is too long and needs dividing up. This is perfectly possible, and is implemented (albeit stepwise) in several editors which use the 'repeated Enter key' method (section 4.3.1.1 on page 233). However, a cæsura of this type, potentially from deep in mixed content markup, all the way up to section or chapter level, is likely to be rare, and can more conveniently be performed by the Split operation.

Again, informed by the principle of 'doing what is expected', any implementation would need to be adaptable to the most common set of behaviours.

4.3.3.2 External material and copy/paste

The drag-and-drop action in graphical user interfaces has been used for many years for moving files between folders and for inserting images into documents. This metaphor has been extended in some interfaces to other functions; the standard method of installing software in OS X (once the image has been mounted) is to drag the application icon onto the desired disk icon; 'Web 2.0' interfaces can now implement a form of drag-and-drop within the browser; dragging highlighted text from one location to another is commonplace in

wordprocessors if not yet in all structured editors; and several interfaces (OS X included) now use the method of dragging an icon off a toolbar or dock into the nothingness of the background or desktop to perform the Remove operation.[18]

This would imply that there is now a well-developed user expectation that this method would also work in a structured-document environment, both for movement between locations and for insertion from external sources. We discuss the editorial function of block movement in section 4.3.5 on page 260, and there is a further note in section 6.3.8 on page 352.

4.3.3.2.1 Images Both XML and LaTeX — being text formats — traditionally use images by reference rather than by embedding, although XML-based office document standards (*OpenOffice*/*Libre Office*'s Office Document Format (ODF) and Microsoft's WordML and Office Open XML (OOXML)) have variously used a combination of both techniques, either embedding a Base64-encoded image in an area of the document reserved for such material and referencing it at the point of usage (WordML), or maintaining the image separately and inserting a link to it in the zip file in the same way as standard XML applications do (OOXML).

The methods widely used in wordprocessors can be used to allow resizing and placement of images, provided facilities exist in the markup for the recording of such information. Where no such facilities exist, images cannot be used without resorting to Processing Instructions or **tag abuse**. Given the low levels of awareness of the concept and facilities of DTDs and Schemas by non-markup-expert authors, a rejected attempt to insert an image into a document of a type that has no facilities for it may become a problem. As with the unavailability of required documents types alluded to earlier, the unavailability of images is perhaps also likely to cause rejection of the software rather than the actual document type, no matter what the justification or rationale is, and regardless of warnings and error messages.

We explain in section 6.3.11.1 on page 356 some requirements for detecting or requesting information about the intended use of certain element types at the time the DTD or Schema is compiled. This can also be used to determine if a particular document type can handle images successfully, or if there are some restrictions that would need to be imposed in the interface, such as limitations on graphics formats or scalability. An image can then be encoded and stored in the

[18] The extreme example in Apple systems of ejecting a disk or Universal Serial Bus (USB) device by dragging its icon to the Trash has been well criticised.

document, if that is provided for; or left on disk and referenced with a filename in an attribute for that purpose; or added via an unparsed entity declaration to the internal DTD subset of the current instance. Similar cases can be made for other media such as audio and video.

In addition to the task of identifying the relevant element for the addition of images and other media, the techniques of Smart Insertion (SI) are also required, to ensure that the element is only inserted in a valid location. In the case of an author or editor dropping an image on the document edit surface, a special cursor can be used to indicate valid locations, but where the process is invoked from a menu, the use of the current text cursor location is implied, and this is where SI is required to identify a suitable alternative.

The problem does not arise in this way with LaTeX documents, as an image can be included almost anywhere (by reference), provided that the enabling package graphicx has been used in the document class file or the author's Preamble (LaTeX's equivalent of the Local DTD Subset); and this is straightforward to undertake programmatically.

4.3.3.2.2 Text In the case of text insertion from external XML sources (as distinct from arbitrary copying and pasting from other windows), it is a requirement of XML that external parsed entities must be integral; that is, they must be well-formed documents in their own right. By implication, they must therefore also be valid with respect to the target DTD or Schema by the time they come to be inserted. Using Target Markup Adoption (TMA), it should be straightforward to insert the contents of such an external file at any valid location indicated by the user, or at a location indicated by the SI method described in section 6.3.2 on page 344. A similar method can be used on LaTeX document fragments, although with less reliability on the point validity for the reasons described in section 1.1.4.2 on page 35.

If the external material is not well-formed, the only option is to strip all markup and insert the content as plaintext, or to request that the errors be fixed before inserting.

If the external material does not use the same markup language as the current document, some form of mapping must exist for TMA to work.

We therefore consider it to be the case that the user will expect drag-and-drop to work seamlessly for all document types, or give good reason why it cannot be

done (for example, attempting to embed a video stream in document being prepared for Braille printing).

We do not intend to test any interface modification for the Add Image function, as the existing menu-and-directory or drag-and-drop methods appear to be adequate, provided there is sufficient markup available to handle them. The key test is for the insertion of marked fragments from other documents.

4.3.4 Formatting controls

Formatting buttons and drop-downs are provided in all but the most elementary interfaces. In wordprocessors, the typeface, font-size, and font-selection buttons operate on any highlighted selection indiscriminately, whereas in structured editing environments the effect is either restricted to the scope of the element content in the element where the highlight starts (most XML editors), or the interface correctly applies separate formatting markup to each span of text, stopping and re-starting at markup boundaries (OOXML and ODF interfaces).

The problems of handling overlapping formatting requirements in a structured environment are well-documented, and are examined briefly in section 1.1.3.2 on page 21: in any case, the formatting applied is generally of the first type discussed in section 4.3.4.1 on the following page, **inline**, which affects only text inside paragraphs, not the paragraphic structure itself. Other formatting buttons, such as those for lists and text alignment, involve structural elements at the paragraph level, and these are dealt with as in section 4.3.4.2 on page 258.

The mechanism for the **differential detection of intent** proposed below, both for the B/I/U buttons and for the typeface, font-size, and text-alignment buttons, is to activate a drop-down menu when clicked, giving a choice of conditions for the formatting (for example, those for italics already mentioned in the list on page 36, including the catch-all option of 'decoration' for 'other purposes'). The behaviour and appearance of the menu is otherwise entirely standard, and when the mouse-button is released, the selected item causes the indicated formatting to be applied. However, in addition, *a*) the relevant markup is added to the document; *b*) the connection between that element type and the chosen formatting is added to the current stylesheet; and *c*) the same formatting can optionally be applied document-wide to all other instances of that element type having the same relative location. The extension of formatting from a single instance to 'all instances of that type' can be handled by a modal dialog option

similar to those in Figure 4.6 on page 238 and Figure 4.7 on page 239 providing the ability to apply the formatting to all similar occurrences.

Meaningful implementation of this requires a specification of precisely which element types in mixed content are to be formatted as bold, italic, underlined, or struck-out; or in a specific typeface or font variant; or ranged left or right or centered or justified. This requires a stylesheet to be constructed to accompany the DTD or Schema, either by hand or as a by-product of DTD/Schema compilation as in section 6.3.11.1 on page 356. Where a user-compiled document type is in use, and no provision is made for element type classification at compile-time, this solution can be used to provide styling for the relevant element type names.

The net effect is identical to the conventional mode of operation in wordprocessor interfaces, with the minor excise of having to move the mouse to select the appropriate meaning, but the process enables the application of both markup and robust formatting in a manner previously inaccessible to the interface.

In each of the foregoing cases, the objective is to allow the writer to contribute to the markup without interference in the conventional act of formatting as you write. There is a strong precedent for using the drop-down **butcon** proposed, in the choices offered by Microsoft *Word* and others in response to pressing the Undo key or menu item (Cooper, Reimann, & David, 2007, p. 500).

During the investigation of this area, colleagues in the Netherlands were kind enough to implement a test interface of the dropdown menu on the [I] button in *Xopus* on foot of suggestions made in an earlier research note (Flynn, 2009). They reported success with their test participants (only one chose the wrong option), but they found that their representation of the *effect* (prefixing the resulting italics with an icon representing the markup) was misleading to the participants (Geers, 2010, §§ 5.2, 6.4.5). In our tests, we do not envisage making use of any method of signalling the markup beyond the formatting and the conventional 'reveal' token in the status line.

4.3.4.1 Inline formatting

This is conventionally limited to the font changes bold and italics, with bold-italic for additional variation. Underlining was originally provided in wordprocessor applications for compatibility with typewriters, where it was the sole form of distinction or emphasis available until the arrival of the IBM *Selectric* ('golf-ball')

typewriter in the early 1960s. It has remained in use, although widely deprecated by typographers, designers, and publishers.

The buttons provided in WYSIWYG interfaces are shown mnemonically as [**B**], [*I*], and [U], possibly with the addition of [~~S~~] for 'strike-through'. Their effect is always **commutative**; that is, they can be used successively, in addition to one another, on the same piece of text, so that the effect accumulates: a bold-italic, underlined phrase can be shown as struck out, ***~~as in this example~~***.

In some technical material, a typeface change from serif to sans-serif, small capitals, or fixed-width type is used where further semantic distinction is required: in drama, small caps are conventional for the names of speakers; and in LaTeX documentation, sans-serif is conventional for the names of packages.

Typeface changes are also commutative with the font changes, so the bold-italic, underlined, struck-out phrase mentioned above can be changed to any typeface in any size available on the user's system. This is perhaps a *locus classicus* of the Art of the Possible: while formatting to that extreme is very rare, other combinations of lesser force are widespread — in order to allow a specific combination of typeface, size, weight, width, slant, and decoration (albeit usually only one or two of them at any time), it is easiest to design a rendering engine to allow *any* combination. While the result achieves the objective of allowing users the freedom to choose combinations of formatting, it also permits them to over-use the facility.

Recording the semantics of such highlighting again involves the differential detection of intent, and a method was needed for this research to capture the relevant information with the minimum of excise (preferably zero).

One such method is to remove the B/I/U buttons and the font controls entirely, and provide dedicated buttons for the most common requirements, especially emphasis. Some near-WYSIWYG editors such as LyX provide additional toolbar buttons for this purpose, as shown in Figure 4.10 on the following page, where *Emphasis* and proper nouns are given their own buttons (the *A!* and following buttons to the right of centre in the top row).

The LyX interface also provides a Font button (to the right of the two just described), and has configuration options for additional variants, but setting up and using them involves some training or learning, and an understanding of the way in which LaTeX handles fonts.



Figure 4.10: The 'standard' and 'extra' tool bars in the LyX editor, showing the 'semantic' approach to typographic variance (*A!* for emphasis and *A* with a human figure for personal names). There are another 20 or so toolbars for specialist purposes.

To avoid this, we continue to observe our predicate of 'minimal change', and retain these buttons, but adapt their behaviour to enable recording of the underlying reason. This is tested with the I button and the original nine possible reasons for italics identified in the list on page 36 (Flynn, 2002).

We have already seen in section 4.3.1.1 on page 228 ff. how a modal dialog can be used to intercept a condition that is known to require attention. This method could be adapted for use with toolbar buttons like I (see Figure 4.11) as well as with the typeface and font-size drop-down menus and the text-alignment buttons, much in the way that an interface will respond to the Ctrl-I, Ctrl-B, and Ctrl-U keystrokes for italics, bold, and underline.

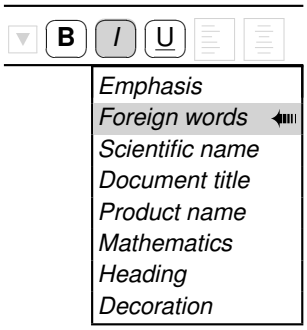


Figure 4.11: Adaptation of the I toolbar font button to drop down a menu

However, typeface selection and font size are less amenable to the detection of intent through a drop-down menu, although the menu itself is technically easily achieved. Most formal documents tend to use a single typeface for the text, possibly with a contrasting one for headings and for out-of-line material such as tables and figures. There is usually little requirement for the author or editor to use either menu provided that the stylesheet is sufficiently comprehensive, because typeface and size tend to be inherited from their containing environment, or specific to a small span of text like the italics used for a cited title. (A possible exception could be the correction of errors caused by faulty retention of formatting when pasting material copied from another window, where TMA has not been implemented.) One option is therefore to do as LyX does, and remove

these menus entirely.

By contrast, a designer or document type controller may spend a good part of the day in using precisely these two controls to manage stylesheets, and the wordprocessor convention follows this pattern by giving relative prominence to them, implying that every author should be able arbitrarily to change typeface and size anywhere in the document at any time.

We need to distinguish here between setting the **document typefaces** (those which the designer applies as standard to normal text, headings, and other contrasting material throughout the document), and **incidental typefaces**, which the user may apply at random anywhere for effect. Size is probably more extensively used at the design stage rather than during actual writing, although from observation there are many authors who will adjust heading sizes to fit certain words or phrases between the margins, or may feel that certain classes of emphasis or key words require larger type.

If an interface continues to provide these menus in a stylesheet-driven environment, and users continue to expect to be able to apply incidental formatting without revealing a consistent reason, the only option is to honour the changes on-screen during editing and draft stages, but to ignore them in the final version. This technique is widely used in publishing production, where authors are allowed to submit *Word* documents using Named Styles from a template, but any incidental formatting of their own devising is routinely ignored, and only the authorised styles used. This somewhat evades the issue (a colleague referred to such toolbar and menu items as 'avoidances', by contrast with affordances), but it is a practicable solution known to work.

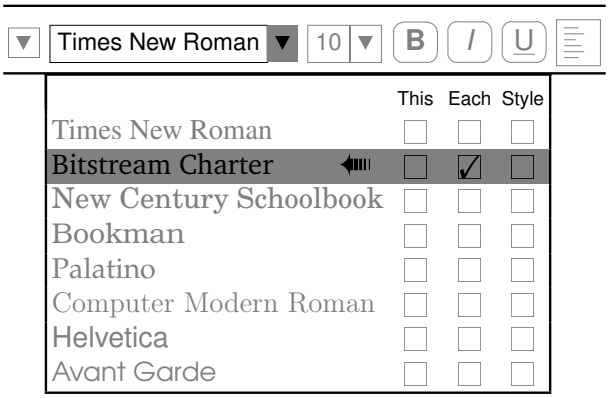


Figure 4.12: Experimental interface adaptation of the typeface menu to record styling requirements

Differential detection of intent is difficult to implement, and therefore to test.

The dropdown area of these menus is already used for the typeface names or sizes, and cannot therefore be used for displaying the list of potential reasons for applying markup, as was possible for the B/I/U buttons.

Figure 4.12 on the previous page shows an experimental implementation (since discarded as being too complex) for recording occurrence information similar to that proposed in the modal dialogs devised for the detection of white-space in section 4.3.1 on page 226. There are checkboxes for applying the selected typeface to the current element only ('This') or to each similarly-placed element ('Each'); a third box ('Style') provides access to the stylesheet in order to make more extensive styling changes (see Figure 3.26 on page 195). However, the space for labelling the selection boxes is limited, and the headings provided in such space are not sufficiently understandable. It is also computationally expensive to provide every typeface in the menu (there may be hundreds installed) with its own set of checkboxes, and it is complex for the user to operate.

Another possible approach would be to allow a right mouseclick or simply a hover (dwell) to bring up a menu of reasons, choices, or other options which would contribute to the markup as well as record the preference in the stylesheet. However, the use of a technique approaching those used in 'Web 2.0' interfaces may be too much excise for what users would regard as a trivial task.

A third possibility would be to retain the conventional menus, but have the program detect the currently prevailing style (based on markup, stylesheet, and the current selection), and ask after a change if the user wants to save the modification or name it as a new style. This has the advantage of minimal excise, and allows the stylesheet to be constructed along the way. However, testing any of these would require a complex and dynamic interface, as the effect would need many more options than are realistically achievable in paper prototyping.

4.3.4.2 Alignment and list formatting

Alignment in text documents usually comes in four orientations: left-aligned ('ragged-right'), right-aligned ('ragged-left'), centered, and justified. In LaTeX, justification is the default, and no separate instruction exists to invoke it. Centering is conventionally used for titling or headings, but is also widely employed purely for æsthetic or decorative purposes.

Left-alignment is most common in informal documents, where the bookish

appearance of justification is not needed or is undesirable. Justification is nevertheless often used in such documents by default.[19] Right-alignment is more rare, but often used as a form of parenthesis to offset quotations, comments, or other ancillary material. Right and left usage is naturally reversed in right-to-left writing systems.

The buttons provided for alignment (visible in Figure 4.13 on the next page) show their orientation in the icons as sets of short horizontal lines representing lines of type. They can be treated for drop-down usage in the same manner as the font buttons (see the example in Figure 4.11 on page 256).

Lists are perhaps the most common structural devices in use after sectioning, and possibly the oldest of which we have evidence, as we saw in section 1.1.1 on page 4. With one exception noted below, they are block structures, traditionally formatted vertically as a series of labelled paragraphs (audio and Braille rendering engines have other ways to identify them).

The numbered-list and bulleted-list buttons in the conventional interface do not require any special treatment like that proposed for the B/I/U buttons, as they already express unambiguously both the intention and thereby the necessary markup. Their operation, when applied to the paragraph in which the cursor is located, or to a selected group of paragraphs, is well-established, creating an item or set of items respectively, in the style specified by the stylesheet.

This process omits a less-frequent but important use: in-paragraph lists, where the formatting is *a*) inline to the containing paragraph; *b*) prefixed by an alphabetic or numeric label and parenthesis; and *c*) separated by semicolons, like this example. This feature is present, if underused, in LaTeX, DocBook, TEI, and HTML, but wordprocessors do not appear to possess anything resembling list markup within paragraphs. Its principal advantages are that it can automate the numbering so that authors do not lose track when editing, and that it can trivially be converted to a vertical list (the final separator token 'and' or 'or' indicates the type of list). In wordprocessor interfaces, so far as can be determined, when a span of text within a paragraph is highlighted and a list button clicked, the entire paragraph becomes a list item, and the highlighting is completely ignored.[20] It

[19] Typographically this is often to their detriment, because the H&J (hyphenation and justification) routines used in wordprocessors are relatively simplistic, and only operate line-by-line, without much optimisation, whereas some typesetting systems consider a whole paragraph at a time, giving a much more acceptable result.

[20] It would perhaps make more sense for such highlighted text in mid-paragraph to be

would seem reasonable that if a list button is applied to a marked span in mid-paragraph, an inline list item should be created if the DTD/Schema permits list markup in mid-paragraph. However, this raises serious difficulties about identifying the bounds of the list, and thus the location for the cursor when the next item is required. It is also insufficiently frequent to justify the depth of attention required for replication in code, and is therefore excluded from the test.

While bulleted and numbered lists have a conventional toolbar icon, there is no common icon for the 'labelled' list (also called a 'discursive' or 'definition' list), where a keyword or phrase is followed by one or more paragraphs of explanation, in the manner of a dictionary (HTML's `dl` and DocBook's `variablelist` element types, or LaTeX's *description* environment). An icon for this type of list was therefore devised and added to the interface, Figure 4.13: following the pattern of the one already in LyX.



The new button for labelled lists is on the right-hand end, after the bulleted-list and numbered-list buttons.

Figure 4.13: Possible instantiation of a toolbar button for a labelled list

4.3.5 Block moves

Moving blocks of text in a wordprocessor involves three actions:

1. highlighting (marking) the text to move;
2. cutting the text from its highlighted position;
3. pasting it into the new location.

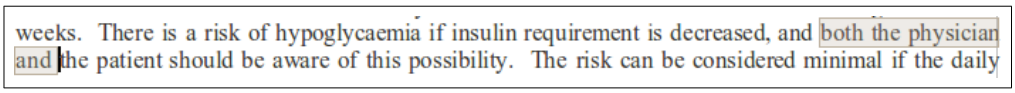
An alternative is for the second two to be collapsed into a single drag-and-drop action. In the WYSIWYG model, no restrictions are placed on what text may be marked or moved: the start-points and end-points can be at any arbitrary location, in accordance with the general principle of permitting the user to do anything anywhere because the software has no knowledge of the document structure and the relative significance of its component parts.

formatted as an isolated list item, inviting either inline or vertical use. The current behaviour breaks the model of performing the action on the highlighted material.

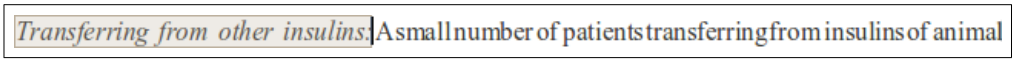
4.3.5.1 Marking for movement

The text which is marked for moving can be described analytically as falling into one of four categories (we use XML terminology here for descriptive purposes):

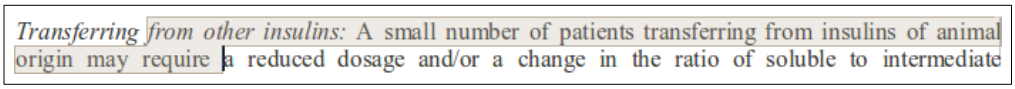
1. **mixed content only**: the start and end of the marked text are within the same element (a paragraph in the example below); any nested subelements are wholly contained within the marked text;



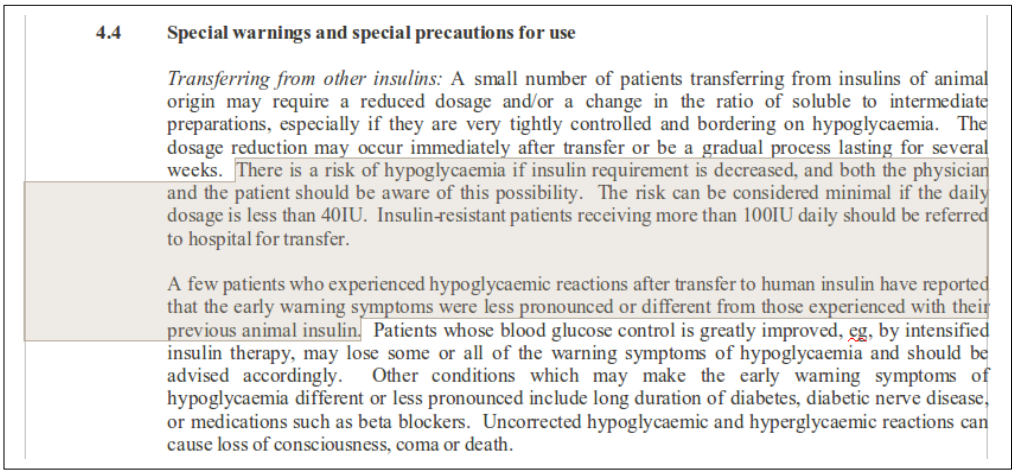
2. **whole element only**: the marked text consists of one or more whole contiguous elements, possibly with subelements within them (this may occur in mixed or element content);



3. **spanned elements in mixed content**: the end-points are located within different contiguous elements, and subelements may also be included;



4. **spanned elements across element and mixed content**: one or both end-points are located within mixed content and the marked text extends across element boundaries in element content



This category would include the case where the marked text starts (or ends) with a whole element in element content, such as a heading, followed by a part of the content of the following paragraph.

The first two cases (mixed content only and whole element only) are straightforward, and the only resolution required is when an element or a nested subelement was valid in the source environment but would not be valid in the target environment. In complex DTDs, it is often the case that some elements are permitted in one content model but not in another, even though they are closely related. DocBook, for example, does not permit lists in an Abstract, so an attempt to paste text containing a list would fail in most editors because it would create an invalid document. The author is aware of a case in an electronic edition of a historical literary work, where a line of verse occurring in direct speech in mid-paragraph (and thus in mixed content) was also required in the title of the document (equally mixed content), but disallowed by the content model (that is, the title element could not contain the element for a line of verse).

Resolution in these cases requires human attention, and it is not clear what can be done to avoid user rejection when the editing software refuses on technical grounds to allow an action that the user either believes to be reasonable or requires *de jure* to be accepted. In the specific case of the line of verse in a title, this was because the designers were unaware that such a circumstance could ever exist, and was simply resolved by changing the DTD; but such swift action is not usually possible in a business or industrial model. Maler and el Andaloussi (1999) explicitly warn against inadequate design processes which fail to admit all required models, and recommend a robust procedure for resolving conflicts.

4.3.5.2 Making the move

The standard implementation of a move operation in the WYSIWYG model simply involves cutting the marked text from the source location, scrolling or otherwise locating the target location, and pasting the contents of the buffer or clipboard. The cut and paste can be done with mouse-clicks, keystrokes, or menu items, or any combination. There are two main problems with large-scale actions, however (those which involve text boundaries off the current screen): scroll acceleration can make it hard to position the mouse to mark the end of the region; and the target location may be nowhere near the source location, meaning a search or extensive scrolling or paging.

Many editors are equipped with a structural navigation panel which displays a tree diagram of the structure of the current document (plug-ins such as outliners may also provide this facility). Typically, nodes in the tree are elements in element

content, and have the conventional plus and minus icons which can be clicked to reveal and hide lower levels of (element) content (see Figure 4.14). These trees are not simply navigation aids, but can be used to move whole elements in element content by dragging-and-dropping them elsewhere in the document tree. In such cases, resolution of content model mismatch is required, as illustrated in section 6.3.1 on page 342. We discuss other related navigation problems in section 4.3.5.3 on page 265.

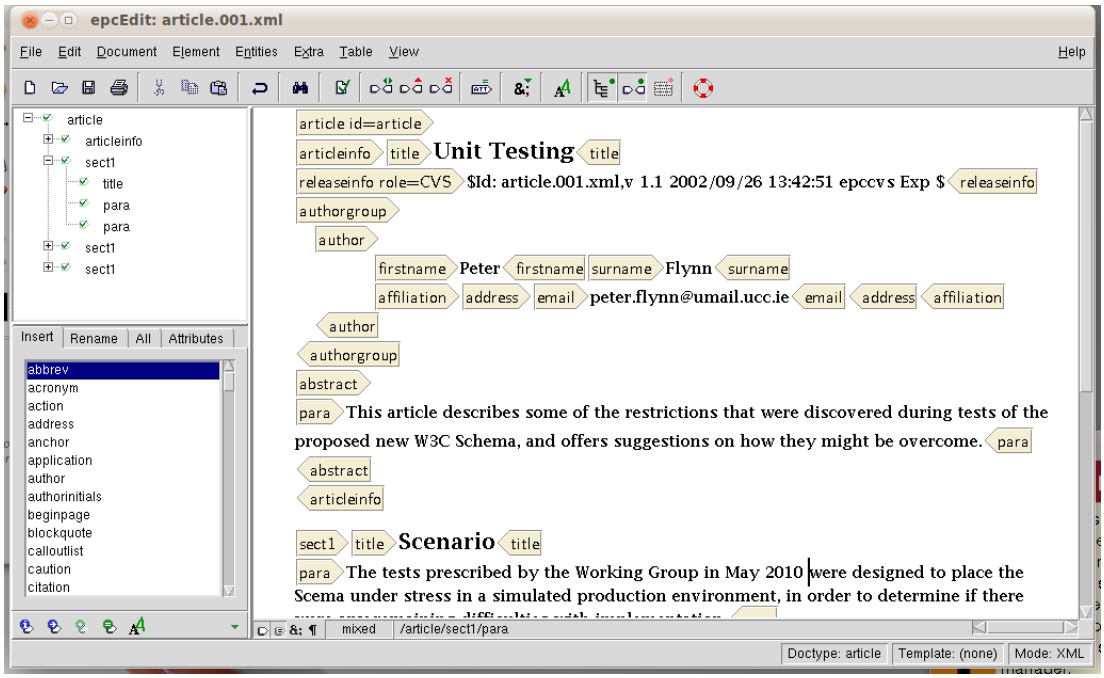


Figure 4.14: Navigation pane in an XML editor

The last two types mentioned in the previous section (spanned elements in mixed content and spanned elements across element and mixed content) pose much more serious technical problems, as they require algorithmic resolution of what to do with the **stubs** (text node or partial element node fragments) at either end of the marked text. Currently the present author is unaware of any editor which offers a solution to this in markup terms (*Emacs* and some other plaintext editors can arbitrarily trespass on markup, but this may invalidate the document).

However, it seems clear that if a text fragment is cut from one end of a paragraph, for example, and an attempt is made to paste it into element content, it should acquire paragraph markup to enclose it, to enable this to happen. Current graphical structured editors will not even allow such stubs to be highlighted, as the moment the cursor moves out of the element in which it started, the entire element gets highlighted.

Where this kind of marked text is cut and pasted into mixed content, there is usually no difficulty in merging the stubs with the preceding or following text nodes, but if there are any wholly nested elements which require element content, the stubs may need to regain their end-of-element role. That is, where a stub has been cut from a subelement in mixed content (possibly followed by text from an intervening span between the subelement and the end of its parent), then it is important to equip that portion of the stub which came from a subelement with another instance of that subelement. This may result in a visual disjunction if the stubs were formatted differently, and may require intervention (such as a new linebreak or the removal of spaces) to preserve appearances. Figure 4.15 shows the result of pasting the marked text illustrated in item 3 on page 261 into another paragraph: note how the wordprocessor has preserved the italics of the first word. In a structured context, this behaviour must also be made possible by creating a new instance of whatever element the word[s] were cut from; or, as a final resort, stripping the markup.

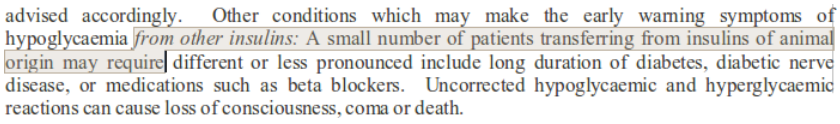

advised accordingly. Other conditions which may make the early warning symptoms of hypoglycaemia *from other insulins:* A small number of patients transferring from insulins of animal origin may require different or less pronounced include long duration of diabetes, diabetic nerve disease, or medications such as beta blockers. Uncorrected hypoglycaemic and hyperglycaemic reactions can cause loss of consciousness, coma or death.

Figure 4.15: Pasted text fragment with stub of overlapped markup

If this is implemented in a structured environment, the editing software must therefore need to take account of the type of element[s] from which the text was cut, and the type[s] into (or between which) it is to be pasted, for two reasons: *a*) the ends of the highlighted text may come from different element environments to the ones into which they are to be pasted; and *b*) any elements wholly contained in the text need to be valid for inclusion into the environment in which the text is to be pasted.

When the text to be moved consists of an integral number of whole elements, with no fragments at the start or end, only the second condition above requires consideration.

If the target location is itself a highlighted span of text, convention is that this second text is *replaced* by the first, which is a second cut operation, and needs to be subject to the same rules for joining stubs as for the first cut.

As mentioned earlier, some editors may forbid the cutting of a selection which extends into stubs at one end or the other (or both); or they may permit it but then forbid pasting it. Most appear to handle this by preventing stubs being

marked: once the mouse moves beyond the end of the element, they extend the start of highlighting right back to the beginning of the containing element in element content; and once the cursor passes into the following element, it immediate highlights all of it. These methods are used to ensure that the only spans which can be selected are *either* whole elements *or* text fragments entirely contained in a single element. The presence of a Paste Special function in the menus may indicate if this is an option.

From a behavioural point of view, however, users are accustomed to the more permissive actions allowed by wordprocessors, and will expect cutting and pasting to 'just work', regardless of any markup constraints (which are in any case largely hidden by the WYSIWYG presentation).

In the process of pasting, a second set of considerations becomes necessary when the target location is not in an equivalent (parallel) position to the source location in respect of markup depth.

TMA, which is described more fully in an implementation note in section 6.3.1 on page 342, provides for the markup in a cut or copied fragment to be aligned with the markup environment of the target location. That is, at the structural level, a level 4 section being pasted into a level 2 section will become a level 3 section in order to be accepted into that location; and inline markup such as HTML's `em` being pasted into a DocBook paragraph can be adapted to become `emphasis`.

Implementation of the mechanics of fragmentary cut and paste in a formal markup environment is not within the scope of paper prototyping, as there are too many alternative ways of working for it to be tractable. Testing is therefore confined to whole-element copy and paste.

4.3.5.3 Navigation

There are several ways to navigate through a document:

1. with the arrow keys, PgUp, PgDn, Home, and End;
2. with special-purpose keystrokes or combinations (for example, *Emacs*' control keys, *vi*'s command-mode keys, or any of the cursor-movement keys defined by successive software vendors, such as the ESDX 'diamond' first used in *WordStar*, named for the position of those keys on the keyboard);
3. with a mouse, stylus, finger, or other movement and pointing device;

4. using a navigation pane, commonly showing a tree-display of the document structure;
5. with an outliner or other system to 'fold away' parts of the document not being edited;
6. using a Search function.

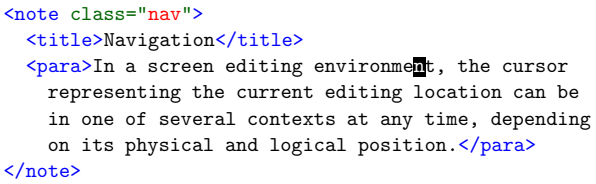
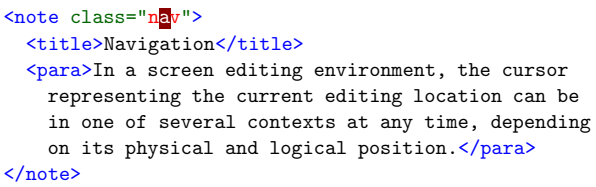
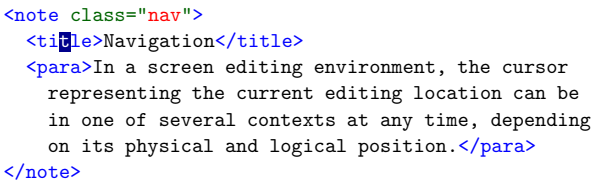
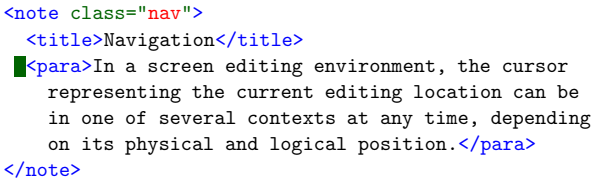
With the first three (traditional keyboard, mouse, or other pointer usage) the cursor representing the current location in a structured document can be in one of several contexts at any time.

In a plaintext editor, with all markup revealed, an I-bar cursor can only ever be *between* two characters, and a block cursor can only ever be *on* a single character. If trespass on the markup is possible, those characters could be *a*) part of the document content (text); *b*) within document markup (tags, entity references, comments, PIs, etc); and *c*) the white-space between the elements in element content, or (in the case of the I-bar cursor), *between* any of these (see Figure 4.16 on the facing page).

By contrast, in a synchronous typographic editor, with all markup hidden, the cursor can only be in the text content, never in the markup (see Figure 4.17 on page 268): *d*) it can be in character data; *e*) it cannot enter tags, entity references, or other markup but must rely on the interface interpreting its location in order to make the markup available for editing; and *f*) in some systems, it cannot be positioned between two elements in element content in order to insert another element — it can only be in character data; that is, immediately before an end-tag or immediately after a start-tag, or in the test between them. This applies also to those plaintext editors which disallow trespass on the markup.

In the non-rigorous environment of a wordprocessor, the problem of inability to place the cursor in element content can be very acute.[21] In a structured editor it is even more essential that the user is allowed access to all those locations where changes might be needed, without allowing the document to be damaged or

[21] A colleague of the present author was using *OpenOffice* to edit contributed chapters for a *Festschrift*, and in deleting the final unwanted endnote of Chapter One, all the rest of the book (twelve chapters) suddenly vanished from the screen. The chapters had been unwittingly and innocently pasted into the last endnote of the first chapter because that was where the cursor happened to be, after editing the first chapter. Inspection showed that there was no way to place the cursor any later in the document structure at that stage, because the end of the final footnote was at the time the last textual object in the document, and there was no access to the markup beyond that point.

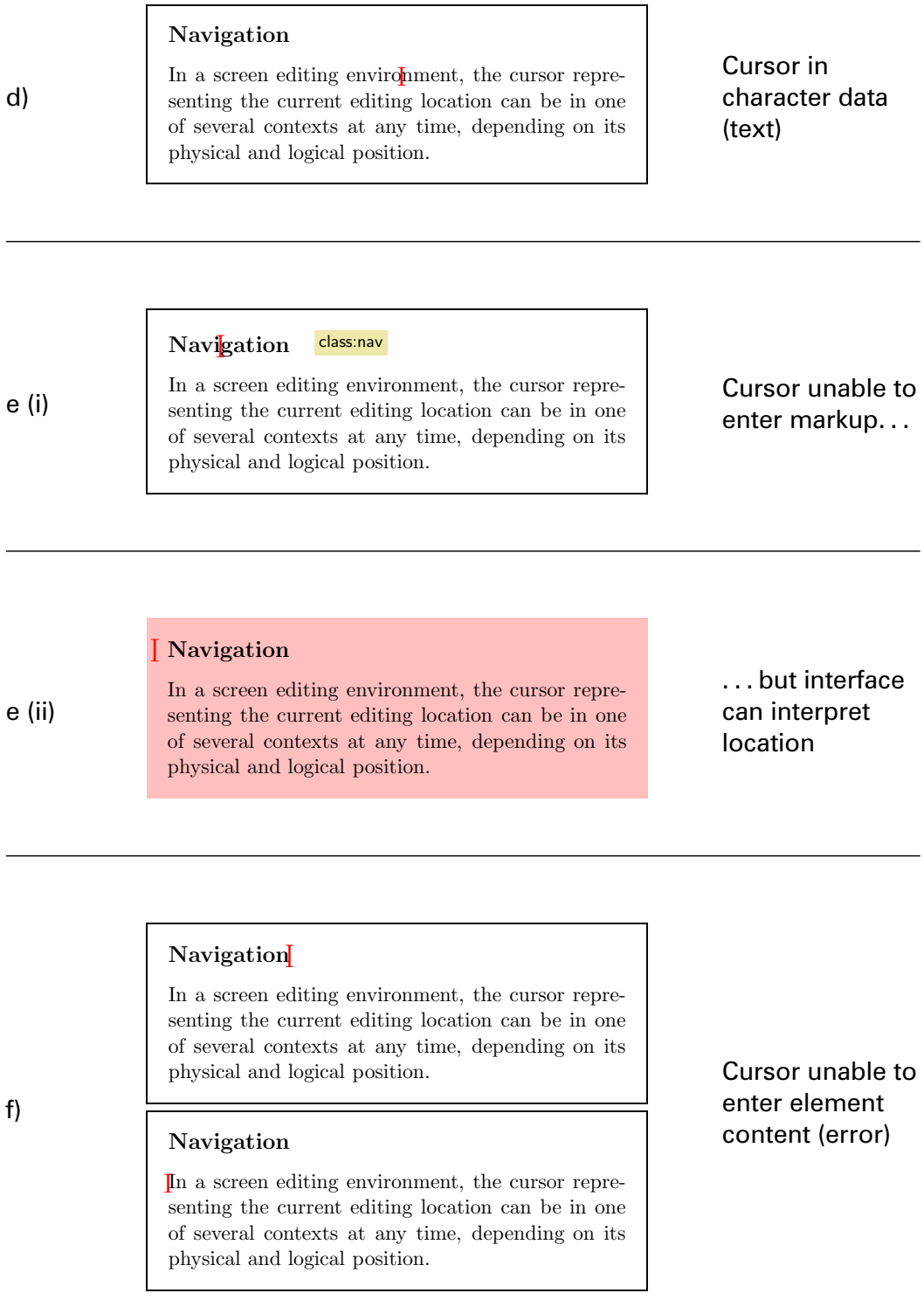
a)		Cursor in character data (text)
b (i)		Cursor in markup (attribute value)
b (ii)		Cursor in markup (element type name)
c)		Cursor in irrelevant white-space between elements in element content

Letters refer to the descriptions on on the preceding page.

Figure 4.16: Some locations of a block cursor in a plain-text XML editor with markup revealed

invalidated.

The difficulties experienced in the example just mentioned could have been avoided if the software had been able to indicate whereabouts in the document the cursor was located, at the time additional chapters were being pasted in; or if it had been possible for the software to detect that the insertions were whole chapters, and thus unlikely to be part of a footnote.



Letters refer to the descriptions on on page 266.

Figure 4.17: Some locations of an I-bar cursor in a synchronous typographic XML editor with markup hidden

Both of these are stock features of XML editors, although less so of LaTeX editors — sensitivity to the markup of pasted or inserted material has already been discussed in detail earlier in this section. A telltale at the foot of the window is now commonplace, giving the element location in breadcrumb format, and allowing the user to navigate to higher sections in the hierarchy by clicking on segments of the breadcrumb. Breadcrumbs are also common on web sites, although in both documents and web pages, it is not a true breadcrumb in the manner of Hänsel and Gretel (which would return you by the path whence you came), but a hierarchical descent indicator (a 'location ladder'[22]) showing where you are, but not how you got there.

4.3.6 Referencing

There are three principal forms of internal cross-reference commonly used in writing:

1. a footnote or endnote, signified in the text by a superscript figure or symbol matching one at the foot of the page or the end of the chapter or the document, where some comment, example or explanation is given;
2. a cross-reference to a location elsewhere in the same document, as 'see Figure 3.2', 'as explained in section 4.7', or 'see footnote 11 on page 42';
3. a bibliographic citation and reference, where the citation is a numeric superscript, a number or contracted author/year abbreviation in brackets, or a full author/year in parentheses; and the reference is a complete identification of the work referred to, printed at the end of the chapter or the document.

In some areas of the Humanities, footnotes and citations are conflated. A superscript citation will refer to a short reference in a footnote, and the full reference is usually given at the end of the document. In some formats, the full reference is given in the footnote on the first citation, and *ibid* used on subsequent occurrences, with no listing of references at the end. As the footnote is used as a place of reference, it is therefore unavailable for comments, examples, or explanations as is conventional in other fields, and the word 'footnote' is often used by scholars in those areas of the Humanities as a synonym for 'bibliographic reference', much to the confusion of colleagues in other areas.

[22]The term comes from the HyTime standard (ISO/IEC 10744:1992, 1992), the Hypermedia/Time-based Structuring Language.

The mechanisms available within XML and LaTeX editing and processing systems for handling cross-references are very similar, although they differ in syntax.

4.3.6.1 Footnotes and endnotes

Footnotes and endnotes are treated here as equivalent: the only difference is in the final placement, which is an implementation detail not relevant to the markup. It is equally not relevant how the footnotes are implemented within the markup: most DTDs (and LaTeX) store them at their point of reference; that is, they are embedded in the text, immediately adjacent to where the footnote mark will be placed (see Figure 4.18).

```
<para>While some editors can be criticised for not providing a
  big enough range of document types (there is a strong emphasis
  on types designed for technical and scientific documents), the
  test is more properly, does the editor provide facilities for
  precompiling and installing new types of document?<footnote>
    <para>The question of whether the user in controlled
      circumstances should be permitted this action is a
      separate matter not addressed in this research.</para>
  </footnote></para>
```

```
While some editors can be criticised for not providing a big
enough range of document types (there is a strong emphasis on
types designed for technical and scientific documents), the
test is more properly, does the editor provide facilities for
precompiling and installing new types of
document?\footnote{The question of whether the user in
  controlled circumstances should be permitted this action
  is a separate matter not addressed in this research.}
```

Figure 4.18: Conventional placement of footnote markup in DocBook and LaTeX

The display of footnotes in the editor differs between editing environments. In a plaintext editor, the display of the footnote is typically shown as in Figure 4.18. In a wordprocessor or other synchronous typographic editor in a page-based display mode, with the page boundary calculated on the basis of the dynamically reformatted text, footnotes can be displayed as they will be printed (small type at the foot of the page). In such cases, the Insert Footnote action causes the cursor to jump to a separate edit area at the foot of the page where the text is entered *in situ*. In a webpage-editing environment, where the concept of fixed-length 'pages'

does not exist, another paradigm must be sought, and the footnote text in the editor may be displayed in some kind of placeholder or subwindow.

Regardless of the method used for display and storage, the invocation of a new footnote is tested here in the New menu; the Insert menu being abandoned for reasons already explained. No change in behaviour is envisaged.

Numbering mechanisms conventionally restart with each new chapter, but may be set to be continuous throughout a document. Where there are very few footnotes, a set of symbols is commonly used, with an asterisk for the first footnote on a page, a dagger for the second, a double-dagger for the third, and so on. These are typically document-wide settings, and unrelated to the interface mechanism of adding a new footnote.

In very complex literary critical works, multiple parallel series of footnotes may be required, for example a numbered series of the original author's original footnotes; a lettered series for the notes of later critics; and a roman-numeral series for the present author's own notes. Such series are kept distinct at the foot of the page, either vertically or horizontally. An interface providing this distinction would need an additional control to specify the series, but that refinement is not tested here. (Other complexities exist in related fields, such as hooked overlines denoting lemmas and readings used in transcriptions or comparative editions, which are equally out of scope here.)

4.3.6.2 Cross-references

In most editing software, an author wishing to refer to another location elsewhere in the document has to ensure that markup to describe the target object already exists before proceeding. In the case of a section in another chapter which has not yet been written, this would require creating an empty section in a skeleton chapter); then edit the label or ID attribute at the target location and assign it a unique name; and finally return to the original location by some means and insert a reference to the newly-identified object.

To the software engineer skilled in XML or LaTeX, the procedure is self-evident: you cannot refer to something until it exists, and so you have to create it and name it before you can refer to it (and the name has to be unique to avoid confusion).

To many writers, however, this is not at all obvious, even when the logic has been thought through. A sufficiently skilled editor will be aware of the necessity, but it

is a tedious task which software should be able to obviate.

One solution, examined here, is that the request for a cross-reference should itself allow the user to scroll or navigate to the target and mark it, and have that action automatically return the cursor to the original point of request and insert the cross-reference, rather the scroll-and-mark action being undertaken separately. If the target does indeed not yet exist, the New menu can be used to create it in the required location beforehand. In the case of a desired target not yet existing (a future, unwritten chapter, for example), the interface should allow the user to signal that fact, and have the editor create the empty target. This refinement is not tested here.

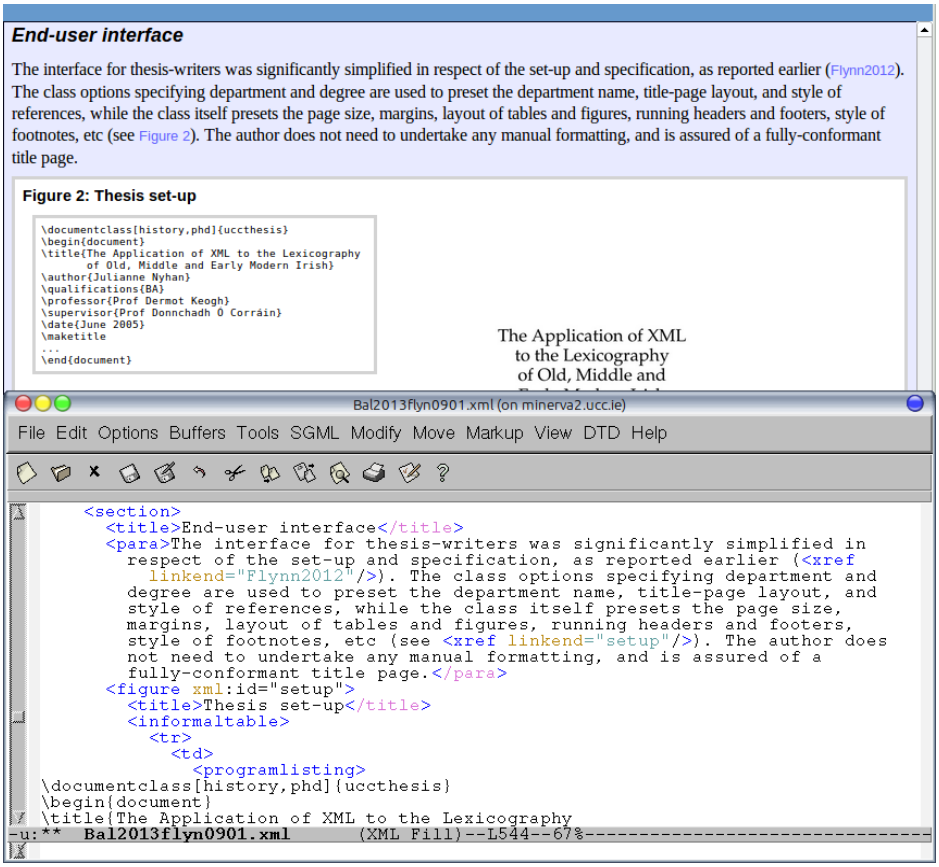
Referring to another part of the same document is commonplace and well-supported in both XML and LaTeX, which have robust mechanisms for dealing with the markup and implementation. Microsoft *Word*, at least in *Word 2010* and greater, uses a scrollable list of 'things that can be referred to' (section headings, list items, etc), which is fast and convenient, but only for that subset of the document: references to other locations still require a target marker to be set first.

In XML, a target element (the **referent**) can be given an attribute of type `ID`, assuming the DTD or schema provides for this; if not, the special `xml:id` type can be used (Marsh, Daniel, & Norman, 2005). The referring element can then use an attribute elsewhere of type `IDREF` or `IDREFS` to refer to it. A validating parser will automatically check that all `IDREF`s match an `ID`. Such `ID` values may occur only once per document, but there may be an unlimited number of `IDREF`s referring to them.

In LaTeX, the referent is identified with a `\label` command, giving a token or label value in the same way as an `ID` in XML. The referring location then uses the `\ref` command with the matching value as the argument, and LaTeX will generate the correct page or sectional number. The rules for occurrence and reference are the same as for XML.

In XML processing, the type of referent can in many cases be deduced from the name of the element type (for example, figure, table, chapter, section, list item, etc) because most DTDs and schemas are written (in English, at least) with explicit names for the element types. But in LaTeX, with less containment, an add-on package called varioref is available, which detects the type of environment or stage in the structure in which a label is located (for example, figure, table, chapter, section, list item, etc), and automatically expands the

reference to add the correct name, so that a reference to the 22nd figure will no longer just be the bare number 22 but 'Figure 22' in full. The package can also add a page number as used in this document. While the XML naming habit is useful, this level of extension has to be implemented in the processing code, such as XSLT, by counting the number of other figures preceding the referent. An example of XML/XSLT's handling of this is shown in Figure 4.19, where the labelling of a figure with an `xml:id` allows the figure to be referred with in an `xref` element, and the reference is completed automatically.



The web-based display of an article during editing shows the resolution of the figure reference (`<xref linkend="setup"/>`) in the formatted text with the 'Figure' automatically prepended.

Figure 4.19: Completion of a cross-reference in an XML system

In an editor, we consider that it is an absolute requirement that full referencing be automatic: in the expectation of the WYSIWYG user, symbolic notations or placeholders are not acceptable. However, the mechanisms of storage still require that a suitably labelled referent be available *before* it can be referred to. In the case of references to earlier parts of the document, this may simply involve the author specifying the referent location with a mouse-click or menu selection. However, during writing, temporary references may need to be made to parts of

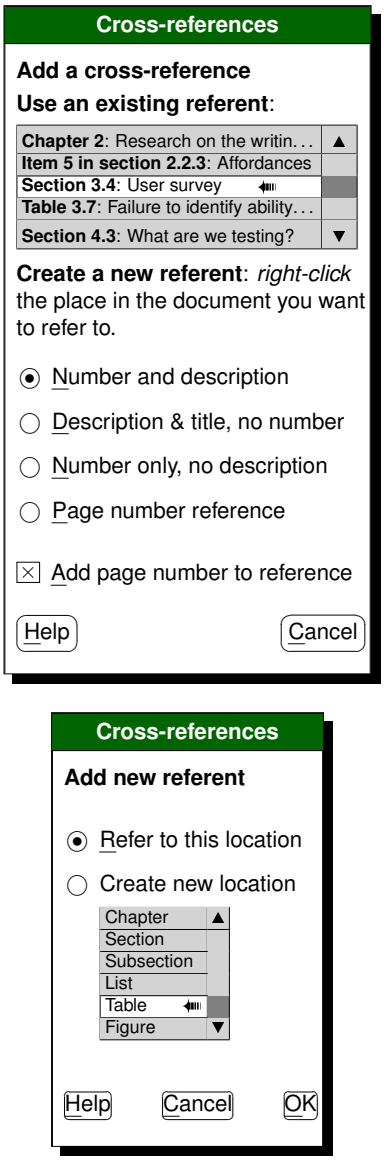
the document that have not yet been written, and for which no referent yet exists. In this case, some kind of skeleton structure needs to be added to the document, to enable the placement of the referent, and to form a platform on which that part of the remainder of the document can be written later.

Wordprocessor solutions to this, while generally accurate and useful, are cumbersome to operate, involving a modal dialog which lacks any understanding of the document structure, and must therefore be configured manually on every occasion (the *Word 2010* interface mentioned earlier is an exception). Mechanisms used in XML and LaTeX editors vary from manual insertion and maintenance of ID labels and IDREF references, to methods using a menu of existing ID values when inserting a cross-reference. When adding a label in LyX, for example, it makes a suggestion for the label value based on an abbreviation of the name of the current environment and the last few words of adjacent text. Modifying such a label after it has been referenced also causes the references to it to be changed in synchrony, but its representation in the display is still a grey box containing the label, rather than a fully resolved reference.

We therefore test a New Cross-reference action which invites the writer either *a*) to select an existing referent from a list (in which case a single click completes the action); or *b*) to identify a new one by scrolling to the location of the desired referent and right-clicking on it (Figure 4.20 on the next page, upper). The list of available existing referents displays their textual context, not their (internal) label value.

If a new referent is required in earlier text, the user can immediately scroll or navigate to the location of the referent because the dialog is modeless and can be made transparent while the document window is being used to navigate to the target location. This avoids the disjunction of apparently completing the operation by clicking OK, when the user knows that the operation is still ongoing (a referent has not yet been selected). When the new referent location is right-clicked (to avoid confusion with normal left-clicks which may be needed during navigation), the lower dialog in Figure 4.20 on the facing page appears: clicking OK sets the referent, dismisses the dialog, returns the user to the original location, fades the original dialog, and adds the reference, formatted as specified. The author is thereby returned directly to the location where the reference was requested, without the need for setting a bookmark or remembering where the original request was issued.

When the referent needs to be set in an as-yet uncreated location, the second



The upper (modeless) dialog appears when the author clicks to add a cross-reference. Selecting an existing one adds it immediately and the dialog is dismissed. Moving away from the dialog to navigate to a new referent location dims or docks the dialog for the duration. On a click at the new referent location, the lower (modal) dialog appears, allowing the selection of an existing element or the creation of a new one (in which latter case the new Add function is invoked, using SI to ensure validity).

Figure 4.20: Possible dialogs for adding a cross-reference

option of the lower dialog in Figure 4.20 is used to create a new instance of the required structure at the point clicked, using the same New method as for insertion. When the dialog is dismissed, the remainder of the operation completes as before. The contents of the list of possible referable elements shown for this second option needs to be populated according to the location where the author clicked to add the new referent. This is an additional argument for the necessity

of some semantic understanding of the markup by the editor. It is also an example of the need to allow element creation beyond the bounds of the current last element containing visible text in the document, as evidenced by the events described in section 4.3.5.3 on page 267.

4.3.6.3 Bibliographic citation and reference

This area is an entire field to itself, with an extensive specialist literature on content, analysis, and formatting. We confine ourselves here strictly to the problems raised in markup by the use of citations and references.

Numerous systems exist for the recording and management of bibliographic references, such as *Endnote*, *ProCite*, *Reference Manager*, *Zotero*, *Mendeley*, *JabRef*, *BIBTEX*, and *biblatex*. Both major wordprocessors also have their own built-in bibliographic databases, although they do not appear to be in extensive use yet; and the TEI, DocBook, and other large-scale DTDs in heavy use offer their own (incompatible!) bibliographic markup. The method of operation is broadly similar for all systems: the author records all the bibliographic data required for each reference in a database distinguishing the component fields such as title, author, journal, publisher, date, volume, issue, etc.[23] The editing software can then add citations to a document, and format the fields of each reference, in the correct manner for the discipline, journal, or publisher, as selected by the author.

Many of the systems available provide wordprocessor and LaTeX users with a way to insert a citation by one of the following means:

- a toolbar or menu item within the editing software which allows selection of the reference from a pop-up window;
- drag-and-drop from the bibliographic reference software's own window into the document;
- double-clicking the relevant reference entry in the bibliographic reference software's own window.

In wordprocessors, the generation of the relevant markup, and the formatting of the display according to the selected bibliographic style is simple and non-problematic. In LaTeX editors, facilities are provided by the *JabRef* program

[23] While this can be done afresh for each document the author writes, it is significantly more effective to accumulate the details of everything relevant as it is read, sometimes over the course of a lifetime.

and others, using one of the methods listed above, because the underlying LaTeX document format provides its own basic set of commands (`\cite` and `\bibitem`) to handle the referencing, and these have been extended by packages to provide for variant ways of making citations.

By contrast, there is currently no standard method for the handling of bibliographic citation and reference in XML apart from the linking mechanism, although numerous tools exist, such as *RefDB* (Hoenicka, 2010), which can maintain and implement a reference and citation database for TEI and DocBook documents.

Many DTDs implement their own conventions, however, and the use of elements using `ID/IDREF` attribute links is commonplace. It then becomes the author's or document engineer's task to arrange for matching `biblioentry` or other element types elsewhere in the document, either using *RefDB* or similar, or as a manual task. A similar position applies to the TEI, which has even more extensive markup for bibliographic material, and numerous ways of creating the citation–reference link.

However, these conventions are unknown to the editing software, although they could be built into an interface: *Arbortext*, for example, provides such a method, as do some DITA editors, and there is customisation available for the TEI using *oXygen*. But there is no generic way for XML software to know what markup or methods are available for an arbitrary DTD, nor any standard way for the DTD or Schema to communicate that information to the program, unless a separate standard like Metadata Object Description Schema (MODS) is used.

It is regarded as essential that identifying this information should be a part of the task of setting up any DTD or Schema capable of bibliographic citation and reference, possibly using the method described in section 6.3.11.1 on page 356 or similar. Where suitable markup is unavailable in a given DTD, it must be assumed that no citations are required, or that they can be supplied unmarked. An alternative would be to use a form of microdata markup such as that offered by Wikipedia or Google, where a visual representation is provided using simple paragraph markup: *Zotero*, for example, already provides pastable references to the clipboard, and already has MODS output available.

In these circumstances, therefore, we test an interface for instantiating bibliographic citation and reference in XML documents similar to that used by wordprocessors and by the *Zotero* plugins for *Word* or *OpenOffice*, driven from

within the editor, rather than the *JabRef* model driven from within the bibliographic database window. This uses a New Citation function based on a toolbar icon (), which switches focus to the bibliographic database window where the document to be cited is selected. No attempt is made to implement a New Reference function, as this is properly the business of the bibliographic database, which could be based on *RefDB* or any other capable product.

In reality it would be be restricted at present to the built-in element types of large systems such as DocBook or TEI, as there is no simple way to generalise this to other document types until semantic identification as described above becomes standard practice. However, as a general principle, it would seem to be preferable to re-use the existing methods already employed in the wordprocessor and LaTeX interfaces, and assume that an external agency is being used to resolve the citation and provide the relevant linking mechanism, in the way suggested for *Zotero* above.

4.3.7 External insertion

Copying and pasting from one document into another is a commonplace and conventional task. In the wordprocessor environment, with an unconstrained document model and only visual means of conveying structure, the cut and pasted selection can be wholly arbitrary. In a structured environment, constraints imposed by the document model mean that (for example) a chapter cannot be pasted into the middle of a paragraph, or — more reasonably — a list item cannot be pasted in between two normal paragraphs.

These constraints could be made to operate in favour of the author: instead of opening the source document, locating the desired text by search or navigation, and marking (selecting) it, a source window could be opened over the target document, showing the source document structure and offering a search (which would present a refined structure representing the search hits). When the desired text is located, it could be copied and pasted by just clicking on it (in the case of a span of character data, it would of course still have to be selected manually).

Because of the 'knowledge' that the system would have of both document structures, a mapping mechanism could be implemented which would largely remove the problem of mismatched structures between the copied fragment and the target location. We discuss some solutions to this in section 6.3.1 on page 342 and section 6.3.2 on page 344.

4.3.8 Editorial assistance

There are several editorial functions which require sensitivity to markup, including spell checking, grammar checking and analysis, outlining, indexing, and content management functions. In the interface, the ability to customise the toolbar can also be regarded as an editorial function. These functions were collected under 'Editorial' in Figure 3.34 on page 202. As we are not testing the behaviour or performance of a spell-check or search itself, any testing of these functions in the interface would have been limited to the existence of an affordance (whether or not there was an obvious button to perform the task). This would not have provided any useful information beyond a positional check on the location of the affordance, and all the software tested which provided the functions already had buttons in the toolbar for them.

4.3.8.1 Spelling and grammar checking

In a structured-document context are both these features are required to operate on the document content only, but to be markup-aware. This means spell checking operates only on the textual material, not on element or command names, or on attribute or option names or values.

However, it can be argued that it should also operate on those attribute values which are normal words or phrases. An example would be the `alt` attribute of HTML's `img` element type, which conventionally contains a textual explanation of the image contents for the use of blind users and those operating without images being displayed. The difficulty is that apart from this well-known and documented instance, there is no clear way for editing software to know which attributes present this type of information. This is another example of the semantics problem discussed in section 6.3.11.1 on page 356, although related to attributes rather than element types.

Markup awareness implies one special condition: when a word *contains* markup, such as syllabic emphasis ('abso*lu*tely'), the intrusion of the interior markup must not split the word and cause fragmented spelling 'errors' (abso , *lu* , and tely, as done in *ispell* or *aspell*), but treat the word as undivided.

4.3.8.2 Searching

This function also requires markup awareness, for similar reasons, but critically it needs the reverse of the condition given above for spelling and grammar, and respect the markup boundaries which terminate mixed content, such as paragraph or section ends.

Users of search engines which do not respect these boundaries (Google, for example) will be familiar with the difficulty exhibited by **markup transgression**, as in searching for a phrase and being given a result which includes one word from the end of a chapter and one word at the start of the title of the next chapter.

4.3.8.3 Toolbar management

In the User Survey analysis in section 3.4.3 on page 189, toolbar use for common functions was not scored particularly highly, which may indicate that it is not used as much for structural editing as for the application of inline formatting (bold, italics, etc), despite the high score on customisation as a recommendable feature (Figure 3.33 on page 201). Methods of implementation vary, and we examine some details of the provision and adjustment of the toolbar in section 6.3.7 on page 351.

4.4 Building the model

A model of editing a structured document is a set of descriptions of the interaction environments in which users find themselves at any particular stage. Implicit in this are a number of pre-existing circumstances or conditions; 'givens' which we must assume exist or have been met. At the physical level there is hardware and software to support the user and the editing environment, and (in the case of editing an existing document) document files to edit. As we are not investigating factors outside the actual editing interface, we shall be taking the external physical factors for granted.

In considering the internal factors, we make explicit the assumptions that will condition and inform the interface which we referred to in section 4.3 on page 222:

1. Adherence to the WYSIWYG model: the results discussed in Chapter 3 made it plain that this is the only model acceptable to the population that we are considering. An editor can obviously provide several different interfaces (plaintext, WYSIWYG, tags-on, etc) to cater for users with other skills or levels of requirement, but the default for the non-markup-expert must be the expected WYSIWYG display. As we established earlier in section 3.1.3.4 on page 122 and elsewhere, the synchronous typographic interface with hidden markup is a principal requirement of any editing interface aimed at the general author or editor. While those with a sound knowledge of markup can write and edit perfectly well (possibly even more efficiently) using a plaintext or graphical interface showing the markup, the effects of nearly 30 years of WYSIWYG interfaces have conditioned the population to expect this as a matter of course in almost any situation where text has to be typed or edited.

2. A *Word*-like appearance. There have been many non-wordprocessor-like editors, both for XML and LaTeX formats, using box models, command interfaces, What You See Is What You Meant (WYSIWYM), floating-menus, and other interface styles (Flynn, 1998). While some have been successful in specialist fields, the prevailing paradigm remains the predominant Microsoft wordprocessor — perhaps ironically, given its recent departure from this convention with the 'ribbon' interface. Whatever failings are attributed by partisans of other systems, it was seen as important for user acceptance to provide this appearance as the default. As with the use of

WYSIWYG above, there is no reason why alternatives should not also be provided for other uses.

3. We predicate the editor upon the Unicode character repertoire. This is not only mandated by XML, but when competently implemented, and provisioned with suitable fonts and substitutions, this ensures that editing can be carried out in any human writing system without the need for extras or plugins such as the TEI Writing System Declarations (WSDs) of the SGML era, which were extremely complex to set up (Birnbaum, Cournane, & Flynn, 1999).

4. Validity should not be a specific concern of the writer. We regard it as the editing software's job to secure, enforce, and preserve the validity of the document at all times, and to explain to the user if a particular combination of circumstances is being disallowed. This does not preclude software from having a separate 'Invalid' mode for experts, where broken files can be repaired, or where testing can be carried out, but such a mode should normally be inaccessible to the user, certainly by accident.

However, at the intellectual level of user capability we are faced with the problem of how, or even if, we should position the editing software to cater for varying degrees of skill and experience. Comments in the User Survey made it clear that 'novice-mode' default settings rapidly become annoying as the new user becomes more fluent in using software, just as 'expert-mode' default settings may prevent the new user from adopting the software at all. We have applied the test of newcomer *vs* expert in respect of the four personas for this purpose (see section 4.2.1 on page 220).

The preconditions derived from the investigations in Chapter 3 could be summarised as including a 'synchronous typographic interface' and that it 'do the right thing'.

The need for the former has been explained above. The latter occurred often in discussions with the participants of the Expert Survey (section 3.1 on page 107), where surprise was expressed that a particular program behaved in a certain way. As experts, the participants understood perfectly at a technical level why certain programs behaved as they did; their surprise was, given the circumstances of daily use, that the programs did not behave in the manner the *users* would expect. Much of this was based on the visual semantics resulting from the actions the interfaces: for example, repeated use of the Enter/Return key in an attempt to

create vertical white-space;[24] and the use of the space or TAB keys to try and make paragraphs indent properly, or combined with forced line-breaks to try and indent a block quotation. In almost all such cases, it was seen either as the 'fault' of the DTD and the stylesheet for failing to provide proper markup and formatting for the type of text concerned; or as the interface's inability to provide the proper affordance for the action. It is important to note that 'doing it right' was in all cases used to mean "doing it right, from the user's point of view", not from the expert's point of view. The case for 'doing it right' is thus to some extent ready-made, as paying attention to the users' point of view forms a cornerstone of the principles of UCD.

In building and testing the model we are therefore concerned with identifying 'the right thing', so that we are matching as closely as possible the expectations of the users with the provisions of the interface.

Finally, as we were not inventing an entire interface from scratch, we only needed to specify the changes we proposed to test in terms of the components we wanted to remove, modify, or add — this is, in effect, a model fragment. For each of these components, we constructed the following:

1. The relevant objective or objectives that were discussed in the foregoing sections;
2. The form that a new or replacement component is to take (the widget);
3. The circumstances of its applicability;
4. The behaviour of the component.

In each test, we then measure the response of the user, record any comments, and derive a metric for the degree of success or failure of the change.

4.5 Testing

Testing was carried out in the Usability Laboratory of the Human Factors Research Group at the School of Applied Psychology, UCC.

[24] In two cases mentioned, the user was attempting to place the following chapter heading at the top of the next page by inserting multiple empty paragraphs immediately above it to fill out the page to the bottom, not realising that the chapter-head markup would do this job automatically. Fortunately, the elision of null paragraphs is trivial to implement.

4.5.1 Test harness

For the experimental tests, a series of screenshots was produced which showed the test document in several conditions, one for the start of each task. From that starting image, there would be three types of response:

1. one or more *expected* responses (keypresses, mouseclicks, cursor movements) which would lead to a known or planned-for result;
2. some which would lead to some other predictable condition;
3. some which would be unexpected, unknown, or not relevant.

A fourth possibility was that the tester would call halt to the task (or the whole test).

The expected responses are shown in section 4.5.4 on page 287. These can be regarded as the canonical responses: those which might be expected from the experienced user who knows how to solve the requirement being tested.

Further images of the document in subsequent states were prepared, taking account of the expected responses. As a result of the three pilot tests, some additional images were prepared to take account of some additional responses which were considered possible occurrences. The image set for each test therefore consisted of:

1. The starting image of the document being edited, in the 'rest position', with the cursor at the desired location.The exceptions were those tests where moving the cursor was itself the action being tested, in which case the cursor was shown in some arbitrary and unrelated position
2. The images of the document after the primary planned responses; that is, the responses expected (from known software behaviour) or allowed for (derived from the foregoing analyses and the pilots);
3. The images of any dialogs or subwindows (for example, navigation) that may be required by the model to result from a response;
4. The final image for each task, showing the document in the situation where the task has been completed.

The full set of 'screenshots' came to 306 separate images (although some were in effect duplicates with only trivial differences such as the position of the cursor. The initial screen for each test is shown in the subsections of section 5.2 on

page 307. The complete set of PDFs are included in the data archive submission accompanying this thesis, along with the LaTeX code which generated them.

A script was written for each task, with rubric to introduce the initial screen and explain the requirement. The script specified the successor sheets to be placed in front of the subject according to the expected responses, and suitable wording to conclude the test. In each case, if a tester called halt to a test, either because they were unable to find a solution, or for some other reason, the final image was not shown.

4.5.2 Test schedule

There were 12 separate tests (tasks):

1. Start a new document;
2. Insert a section from another document;
3. Add new paragraph after the current one;
4. Split the current paragraph;
5. Join the current paragraph to the preceding one;
6. Join the current paragraph to the following one;
7. Add a new section to the document;
8. Make a word or phrase italic or bold;
9. Add a new list to the document;
10. Mark and move a chunk to another location;
11. Add a cross-reference to somewhere else in the document;
12. Add a citation.

4.5.3 Rubric

The introduction for testers was read to all participants (see reproductions in appendix section D.1 on p. 404):

Testing changes to an editing interface

We are testing some changes to the way document editors and wordprocessors work. The program you will be testing is much like any conventional document editor or wordprocessor, but there are a few new icons in the toolbar, and a couple of new menus. Before we start the test, you will be able to look at the screen so that you can familiarise yourself with the menus and toolbars.

The method we are using is called 'Paper Prototyping', where the screen is replaced with printed screenshots. You can point and 'click' with your finger, and the 'Computer' (a person!) will replace the printout with another one showing the result of your click. No typing is involved, but we have provided a keyboard so that you can press real keys if you need to, instead of telling the Computer what the keys would be.

You will be asked to test twelve tasks, each of which involves making changes to a document, exactly as if you were writing it yourself. The example document happens to be a magazine article about software piracy, but that isn't really relevant: it could be about anything in your field of expertise.

Each task to be tested takes a few minutes, often less. There are several different ways to do each task, all equally correct. What we are measuring is how effective the program would be at doing the tasks, and how obvious it would be to an author how to do them.

Each task starts with a short explanation of what is needed, and the Computer will then show you the starting screen. When you have decided what button to click or when menu to use, point at it as if your finger was the mouse cursor, or tell the Computer the name of what you would click on. The Computer will then show you the screen that would result from your choice. You can change your mind at any time and go back to the previous screen by asking the Computer to go back (as this isn't a browser, there is no Back button). You can get rid of a menu in the normal way by clicking in an empty area of the window.

Please remember that there is no right or wrong in these tests: we are not measuring you, we are measuring the program represented by the screens.

4.5.4 Expected responses

The actions expected for each task are listed here in the following subsections with an explanation of alternative paths which a tester may take.

4.5.4.1 Start a new document

Traditional path File | New, possibly with document type offerings.

Via the New menu New | Document | Article

Via the New-Doc toolbar icon ( ) New-Doc | Article

Via the New toolbar icon ( ) New | Article

Control File | New | Article

4.5.4.2 Insert a fragment from another document

Traditional path Move to location; File | Open; select file; navigate to location; Mark toolbar icon ( ); move; Copy toolbar icon ( ); switch window; Paste toolbar icon ( )

Control Move to location; Insert | Document or fragment; select document; select fragment

4.5.4.3 Add new paragraph after the current one

Traditional path Move to location; Enter; Enter

Via the New menu New | Paragraph

Via the Insert menu Insert | Paragraph break

Via the New toolbar icon ( ) New | Paragraph

Control Move to location; Enter (second Enter is caught by modal dialog)

4.5.4.4 Split the current paragraph

Traditional path Move to location; Enter Enter

Via the Edit menu Edit | Split

Via the Insert menu Insert | Paragraph break

Potential error New | Paragraph creates new paragraph after the current one

Control Move to location; Enter (second Enter is caught by modal dialog)

4.5.4.5 Join the current paragraph to the preceding one

Traditional path Move to start of paragraph; Backspace

Alternative traditional path Move to end of preceding paragraph; Del

Via the Edit menu Move to start of paragraph; Edit | Join to preceding

Via the Edit menu Leave the cursor where it is; Edit | Join to preceding

Control As traditional (either)

4.5.4.6 Join the current paragraph to the following one

Traditional path Move to end of paragraph; Del

Alternative traditional path Move to start of following paragraph; Backspace

Via the Edit menu Move to end of paragraph; Edit | Join to following

Via the Edit menu Leave cursor where it is; Edit | Join to following

Control As traditional (either)

4.5.4.7 Add a new section to the document

Traditional path Move to location; Enter; Enter; type heading; adjust formatting

4.5.4.8 Moving the cursor to the end

Via the New menu New | Section

Via the New toolbar icon () New | Section

Control Move to location; Enter Enter; second Enter is caught by modal dialog

4.5.4.9 Keeping the cursor where it is

Control is not needed on this route, as this is a break from the traditional path.

Via the New menu New | Section

Via the New toolbar icon () New | Section

4.5.4.10 Make a word or phrase italic or bold

Traditional path Mark the text; I toolbar icon ()

Alternative traditional path Ctrl I at start; Ctrl I at end

Via the Format menu Highlight the text; Format | Product name

Via the I toolbar icon () Highlight the text; Format | Product name

Control Mark the text; I | Product name; if Ctrl-I is used, the I toolbar icon () should activate on the first and display the menu on the second

4.5.4.11 Add a new numbered list to the document

Traditional path Move to location; Enter Enter 1.SP gets detected as start of numbered list

Alternative traditional path Move to location; Enter Enter and start new paragraph; after a sentence or so, invoke numbering from the menus or toolbar icons.

Via the New menu New | Numbered list

Control Create new paragraph then invoke numbering from toolbar icon

4.5.4.12 Mark and move an integral block of text to another location

Traditional path Mark; Edit | Cut (or the Cut button); move/find; Paste

Via the Mark and Outline buttons Locate block; Mark; Outline; click location; Move marked

Control Traditional cut and paste

4.5.4.13 Add a cross-reference to somewhere else in the document

Traditional path New method in *Word* using pop-up navigation panel

Via the Insert menu Insert | Cross-reference | Insert reference; search or click; click on Refer

Via the Crossref button Crossref; search or click; click on Refer

Control As for non-*Word* wordprocessors

4.5.4.14 Add a citation

You want to add a citation at the end of the first sentence of Section 4, where it says "... original manufacturer's packaging." The reference you want is a book by Paul Paradise called *Trademark counterfeiting, product piracy, and the billion dollar threat to the U.S. economy* and it is already in your database of references. Your cursor is at the point where you want the citation to appear.

Traditional path Using external or internal bibliographic database

Via the Insert menu Insert | Citation; click to select

Via the Cite button Cite; click to select

Control mimic in Word

4.5.5 Administration

4.5.5.1 Selection of testers

The original design was to have twenty testers, with five each from the fields of Business, Academia, Government, and NGOs. The requirements were:

- familiarity with using a wordprocessor or other editing program;
- work involves (occasional or frequent) writing of documents which have a known structure (for example, reports, articles, essays, white papers, etc);
- no expertise in structured document techniques or languages (familiarity is no bar, but expertise is undesirable);
- must not be a professional writer, documenter, or documentation engineer.

The population of potential testers was explored by discussion with colleagues in the fields described, who were asked if they could identify candidates who fulfilled the criteria above. It became clear that asking candidates to request voluntary, unpaid absence from their daytime job, to travel across the city, and attend a testing session (in all, perhaps 2–3 hours away from work), was an unreasonable request, and unlikely to be successful. A test questionnaire to determine experience was piloted and announced on local mailing lists and Twitter, but attracted only one (invalid) response.

Instead, a request for volunteers was posted internally within the university, using mailing lists for academic staff, administrators, professional technical staff, and postgraduate students. While this approach is not ideal, the inclusion of administrators partly compensates for the absence of business users, as their document requirements would be relatively similar (that is, documents relating to the administration of an organisation, not academic or research texts). Professional technical staff in fact bring an additional facet to the requirements which was not present in the original design. Postgraduate students have similar writing requirements to full-time academics, but would represent a much younger demographic to the panel, possibly with different sets of expectations to their older colleagues in relation to their use of interfaces.

As with the pilot mentioned above, the request for participation asked volunteers to signal their candidacy via a short online questionnaire. This was modified to reflect the institution's population, and to collect sufficient information to allow us to detect anyone not fitting the profile items listed above. No mention was made of the profile requirements, so as not to influence volunteers' candidacy. The questionnaire asked for the following items:

- Occupation (for example, academic, student, etc);
- Types of document written (books, articles, reports, etc), and quantity written per year;
- Software used (for example, Word, LaTeX, etc);
- Area of work or study (arts, engineering, sciences, etc).

A full copy of the questionnaire is in appendix D on p. 399.

The questionnaire was left open for two weeks and gathered 61 responses, of which 40 were incomplete (subject decided not to complete it after all). The profile of the valid responses is given in Table 4.2 on the following page

to Table 4.4 on the facing page and Figure 4.21 on page 294.

Table 4.2: Responses to request for testers: Number of candidates by occupation and discipline

		Occupation ⟶					
	Total Sample	**Academic**	**Research**	**Support**	**Post-graduate**	**Technical**	**Retired**
Total Sample	**21** 100.0%	**6** 100.0%	**3** 100.0%	**2** 100.0%	**8** 100.0%	**1** 100.0%	**1** 100.0%
Engineering	3 14.3%	1 16.7%	–	–	2 25.0%	–	–
Computing	2 9.5%	–	–	–	1 12.5%	1 100.0%	–
Humanities	7 33.3%	2 33.3%	–	2 100.0%	2 25.0%	–	1 100.0%
Natural Sciences	4 19.0%	2 33.3%	–	–	2 25.0%	–	–
Social Sciences	6 28.6%	2 33.3%	2 66.7%	–	2 25.0%	–	–
Medical	3 14.3%	–	1 33.3%	–	1 12.5%	–	1 100.0%
Missing	1 4.8%	–	1 33.3%	–	–	–	–
Base	20 95.2%	6 100.0%	2 66.7%	2 100.0%	8 100.0%	1 100.0%	1 100.0%
Responses	25	7	3	2	10	1	2

The spread between disciplines appeared reasonable, and there was at least one support and one technical candidate.

As expected, *Word* was by far the most common system used (see Table 4.3 on the facing page). The presence of Google Docs was a welcome addition. Although its functionality is limited compared with non-cloud systems, it has a functioning stylesheet mechanism, and can export to a variety of formats.

In the types of document shown in Table 4.4 on the next page, the Fiction/Other group included plays, poems, and short stories. Structured authoring, as we have shown, is not restricted to technical documents.

The number of documents reported to be written per year included an entry from two candidates who had included Email as one of their outputs, and had calculated that they sent between 1,000 and 2,000 a year, whence the final column in Figure 4.21 on page 294.

The 21 candidates from these responses were then contacted by email, explaining again the nature of the research, and asking again if they would be willing to help by testing some new software features. No further evaluation or discussion took place except to explain more about the test, if asked.

Table 4.3: Responses to request for testers: Number of candidates by occupation and software used

	Occupation ⟶					
	Total Sample	**Academic**	**Research**	**Support**	**Post-graduate**	**Technical**	**Retired**
Total Sample (multiple-choice)	**21** **100.0%**	**6** **100.0%**	**3** **100.0%**	**2** **100.0%**	**8** **100.0%**	**1** **100.0%**	**1** **100.0%**
Microsoft Word	19 90.5%	6 100.0%	3 100.0%	2 100.0%	6 75.0%	1 100.0%	1 100.0%
Google Docs	2 9.5%	1 16.7%	–	–	–	1 100.0%	–
LaTeX	4 19.0%	1 16.7%	–	–	3 37.5%	–	–
Missing	–	–	–	–	–	–	–
Base	21 100.0%	6 100.0%	3 100.0%	2 100.0%	8 100.0%	1 100.0%	1 100.0%
Responses	25	8	3	2	9	2	1

A Doodle poll was created and circulated to all candidates, with the candidates able to select one slot (and each slot limited to one participant).

4.5.5.2 Pilot tests

Before finalising the screen images and rubrics, three pilot tests were conducted with colleagues from the author's research group. This was partly to utilise their

Table 4.4: Responses to request for testers: Number of candidates by occupation and types of document

	Occupation ⟶					
	Total Sample	**Academic**	**Research**	**Support**	**Post-graduate**	**Technical**	**Retired**
Total Sample (multiple-choice)	**50** **100.0%**	**15** **100.0%**	**8** **100.0%**	**3** **100.0%**	**20** **100.0%**	**2** **100.0%**	**2** **100.0%**
Books	4 8.0%	3 20.0%	–	–	–	–	1 50.0%
Articles	16 32.0%	5 33.3%	3 37.5%	–	7 35.0%	1 50.0%	–
Fiction/Other	4 8.0%	1 6.7%	–	1 33.3%	1 5.0%	–	1 50.0%
Reports	13 26.0%	2 13.3%	3 37.5%	2 66.7%	5 25.0%	1 50.0%	–
Theses	13 26.0%	4 26.7%	2 25.0%	–	7 35.0%	–	–
Missing	–	–	–	–	–	–	–
Base	50 100.0%	15 100.0%	8 100.0%	3 100.0%	20 100.0%	2 100.0%	2 100.0%
Responses	50	15	8	3	20	2	2

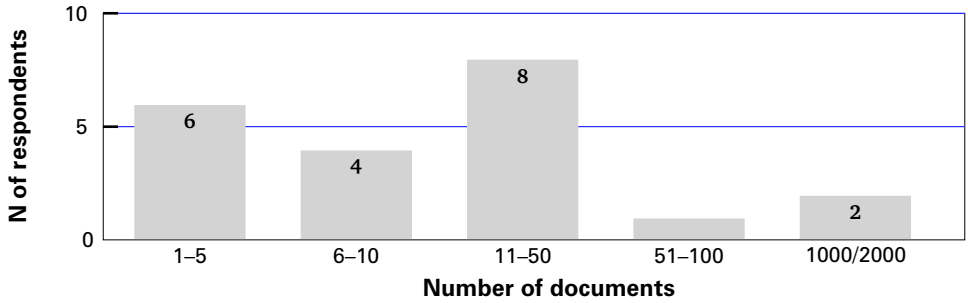


Figure 4.21: Responses to request for testers: annual number of documents written

existing expertise in conducting laboratory usability tests, and partly because — as academics with writing experience — they fulfilled the criteria in section 4.5.5.1 on page 290.

These pilots revealed several typographic and semantic errors which were easily corrected, and five errors in the chain of screen images being used that did not provide for the use of some keystrokes. These were also corrected by generating the additional screens required.

4.5.5.3 Protocol

Candidates were emailed a reminder of the date and time they had selected, with details of the location.

As the testers were all adults participating voluntarily, with full information provided, a guarantee of anonymity, and no personal information involved, it was not necessary to use a signed consent form.

On arrival at the test location, each tester was met and welcomed, and conducted to the testing laboratory. The nature of the research was explained again, and the environment of the laboratory, computer, and paper prototyping method was explained, including the existence of the dark-glass internal window and the possibility of observers being present in the adjacent observation room. Confirmation was given that no recording or video was being made, only a paper record of the responses, which was in plain sight. Verbal agreement was obtained again before proceeding.

At the start of the test, each tester was taken through the explanatory sheet (see appendix section D.1 on p. 404), and asked if they understood what was required. Additional explanation was given if requested.

The tasks were presented in order, using the scripts shown in appendix section D.2

on p. 408. Keystrokes and mouseclicks by the user were recorded on the sheets, and any divergences from the expected patterns were noted in the space provided.

When the tests had finished, the tester was asked if they had any comments or suggestions, using the three questions shown on p. 425. These were discussed briefly, and the tester was thanked for participating.

Each tester was presented with a small token of appreciation for their time and effort (a chocolate keyboard) on conclusion of their session.

Chapter 5

Results

1. Data processing — Computation — Analysis. 2. Individual test results — Test 1: Create a new Article Journal document — Test 2: Add a new paragraph after the current one — Test 3: Split a paragraph into two — Test 4: Join a paragraph to the preceding one — Test 5: Join a paragraph to the following one — Test 6: Add a new section to the article — Test 7: Adding a new list — Test 8: Move a block of text from one place to another — Test 9: Highlighting a trade name (product name) — Test 10: Add a cross-reference to another section — Test 11: Give a citation to an article you need to refer to — Test 12: Insert a fragment of another document. 3. Divergences from patterns in tested behaviour. 4. Comments and discussions — Informal responses — Criticisms.

Twenty-one testers participated during June and July 2013. The results of their tests were transcribed into a data file and prepared for analysis.

No major problems were encountered during testing. A minor mistyping was noticed in the rubric to Task 3 (‘after’ instead of ‘before’). This was corrected verbally in the first test, and subsequent sheets were reprinted. As noted in section 5.2.8 on page 319, the Move button was omitted from the toolbar, but this only potentially affected three testers in one test who did not use cursor highlighting, and was not actually sought by any of them.

A few significant divergences from the expected patterns were observed and recorded while testing. These appeared to be related mostly to the concept of splitting a paragraph, and are discussed in section 5.3 on page 329.

Individual tasks were not timed, but each session (one tester doing 12 tasks, plus

discussion) took about 30–40 minutes, which accorded with the time expected based on the pilot tests.

5.1 Data processing

The data was transcribed into a CSV file as shown in Figure 5.1. The first field is the tester ID; the second is the task number (1–12 for the tasks, 13–15 for the subsequent questions); and the remaining fields on each line are one-word mnemonics for the keystrokes or mouseclicks used in each task.[1]

```
4,1,newmenu,document,articlej
4,2,cursor,newbut,para
4,3,cursor,enter
4,4,cursor,delete
4,5,cursor,delete
4,6,cursor,newmenu,section
4,7,cursor,enter,numlistbut
4,8,mark,hilite,cursor,cut,cursor,enter,paste
4,9,hilite,ctli,prod
4,10,xref,copyright
4,11,cite,paradise
4,12,cursor,insert,docfrag,yourname
4,13,very
4,14,fewer,TeXnicCenter
4,15,yes
```

Figure 5.1: Format of transcribed data (case 4)

The patterns (potential solutions) from the test recording sheets (see appendix section D.2 on p. 408) were created in a similar file layout as shown in Figure 5.2 on the next page. The first field is the number of the task (1–12); the second is the ranking of the pattern; the third is an 'affordance class' (new, old, or both), and the remaining fields on each line are the mnemonics.

The classification of the patterns by affordance is done to overcome the stepwise nature of the rank index (more on this in section 5.1.2 on page 305). The mnemonics used are a controlled vocabulary for the (limited) set of keystrokes or

[1] The mnemonics were chosen for brevity and ease of processing. The rules for naming consistency, such as those first proposed in Green and Payne (1984), which would be useful in designing a task language, were sacrificed on the basis that the descriptors used here are by definition atomic, and are only related for the duration of the tests.

```
1,1,n,newdoc,articlej
1,2,n,newbut,document,articlej
1,3,n,newmenu,document,articlej
1,4,o,filemenu,new,articlej
2,1,n,newbut,para
2,2,n,newmenu,para
2,3,o,cursor,enter
2,4,on,cursor,newbut,para
2,5,on,cursor,newmenu,para
```

Figure 5.2: Format of transcribed patterns (tasks 1 and 2)

mouseclicks used in these tests. They are context-sensitive, following exactly the rubric of the recording sheets in appendix section D.2 on p. 408, so for example in Task 4 (‘Join a paragraph to the preceding one’), `cursor,backspace` means ‘move the cursor to the start of the paragraph indicated and press Backspace’; whereas in Task 5 (‘Join a paragraph to the following one’), `cursor,backspace` means ‘move the cursor to the start of the following paragraph and press Backspace’.

The ranking (second field in the patterns) is important, as it is used determine the overall scores computed for each case:

- Lower rank values represent those patterns which describe solutions to the task *using one or more of the changes to the interface which we are testing*;
- Higher rank values represent those patterns which describe solutions to the task *in terms of existing methods*.

The division between these descriptions is not exact, because some of the patterns include both new and existing solutions to the task. For example, in Task 1, rank 1 is `newdoc,articlej`, which means the New-Doc toolbar icon ( ) icon followed by the Journal Article menu entry. This is the solution using the toolbar icon and menu item we provided, hence the rank 1. Task 1, rank 4 is `filemenu,new,articlej`, which means the File | New menu item followed by the Journal Article menu entry: this is the solution using an existing method (the File | New menu), hence the larger rank value. Task 2, rank 5, however, is `cursor,newmenu,para`, which combines an existing cursor-positioning move with the use of a new feature. Given the monodic nature of ranking, the order has consistency: all-new followed by all-existing followed by hybrid. While this is not ideal, it is at least consistent, and therefore comparable.

5.1.1 Computation

Processing was carried out using the *P-Stat* package used in earlier work (Buhler & Buhler, 1990). The final three questions for each tester (the qualitative questions 13, 14, and 15) were removed from the data and stored separately for later analysis (see section 5.4 on page 332).

This left 12 task measurements for each tester, in the form of a variable number of single-mnemonic tokens (variables named `K.01` upwards; see Figure 5.3). Ten keystrokes/mouseclicks were allowed for; in fact the maximum needed was eight.

Tester	Task	K.01	K.02	K.03	K.04	K.05	K.06	K.07
4	1	newmenu	document	articlej	-	-	-	-
4	2	cursor	newbut	para	-	-	-	-
4	3	cursor	enter	-	-	-	-	-
4	4	cursor	delete	-	-	-	-	-
4	5	cursor	delete	-	-	-	-	-
4	6	cursor	newmenu	section	-	-	-	-
4	7	cursor	enter	numlistbut	-	-	-	-
4	8	mark	hilite	cursor	cut	cursor	enter	paste
4	9	hilite	ctli	prod	-	-	-	-
4	10	xref	copyright	-	-	-	-	-
4	11	cite	paradise	-	-	-	-	-
4	12	cursor	insert	docfrag	yourname	-	-	-

Figure 5.3: Test data representation (case 4)

The pattern data (which is what the measurements will be compared with) was stored the same way (variables `P{}.*`; see Figure 5.4).

Task	Pattern	P.01	P.02	P.03
2	1	newbut	para	-
2	2	newmenu	para	-
2	3	cursor	enter	-
2	4	cursor	newbut	para
2	5	cursor	newmenu	para

Figure 5.4: Pattern data representation (Task 2)

As there was a variable number of possible patterns (solutions) for each task, all of the `P{}.*` patterns for each task for each tester had to be compared with all the `K.*` measurement variables so that the actual pattern used (if any) could be

identified.[2]

Tester	Task	K.01	K.02	K.03	P.01	P.02	P.03
4	2	cursor	newbut	para	newbut	para	-
4	2	cursor	newbut	para	newmenu	para	-
4	2	cursor	newbut	para	cursor	enter	-
4	2	cursor	newbut	para	cursor	newbut	para
4	2	cursor	newbut	para	cursor	newmenu	para

Figure 5.5: Test and pattern data arranged for matching

Each task record was duplicated as many times as there were patterns for the task, and one pattern joined to the end of each such record, so that the patterns could be stepped through in synchrony with the test measurements. In the example in Figure 5.5, the measurement for Task 2 for Tester 4 has been lined up with the five patterns for that task. The closest match (in fact an exact match) is in the fourth line of the data shown.

For each record, the procedure can now test each `K.*` token against each pattern token. The matching was done by stepping through the values for the `P{}.*` variables in order, and testing them against the `K.*` values recorded for a task, using independent pointers p and k (for the program code, see appendix section C.2.5 on p. 396).

If a token matched, both the pattern and the task pointers were incremented to the next position. If a token did not match, only the task pointer k was incremented. This method has the effect of 'marking time' on the pattern variable while a task variable remains unmatched, to see if the next task variable might match instead, allowing us to skip over false moves or divergences. A mismatch counter m kept track of how many times this skip occurred. When either pattern or task data ran out, the loop was terminated and the values of the pointers and the mismatch counter were stored as extra variables on the case.

Finally, the data was sorted into ascending order of the number of mismatches for each tester and task, so that the lowest value for any one case can be taken as the closest matching occurrence between task data and pattern data. This therefore represents the solution for that task for that tester. For example, in the case of

[2] I am indebted to Roald and Shirrell Buhler of P-Stat, Inc for the efficient solution to this (in appendix section C.2.4 on p. 395). It is with sadness that I must record that Roald died (November 2013) before this report was completed, but he had the satisfaction of knowing that this method worked.

Tester 4 on Task 2 (from Figure 5.3 on page 300 and Figure 5.4 on page 300), we get an exact match $p = k$ for pattern 4, and no divergences ($m = 0$):

Tester	Task	Pattern	p	k	m	K.01	K.02	K.03
4	2	4	3	3	0	cursor	newbut	para

By contrast, Tester 8 on the same task pressed the Enter key a second time (and would have received and dismissed the multiple-Enter dialogue; see Figure 4.4 on page 234), so there was one divergence:

Tester	Task	Pattern	p	k	m	K.01	K.02	K.03
8	2	3	2	2	1	cursor	enter	enter

This provided a clean data set giving the pattern (solution) which was used by each tester for each task, how many keystrokes or mouseclicks were needed, and how far (if at all) the tester diverged from expected patterns to achieve the task.

5.1.2 Analysis

All testers completed all of the tasks successfully with one minor exception, although there were numerous divergences from the expected patterns *en route*. The distribution of these divergences is shown in Figure 5.6 and is discussed in more detail in section 5.3 on page 329.

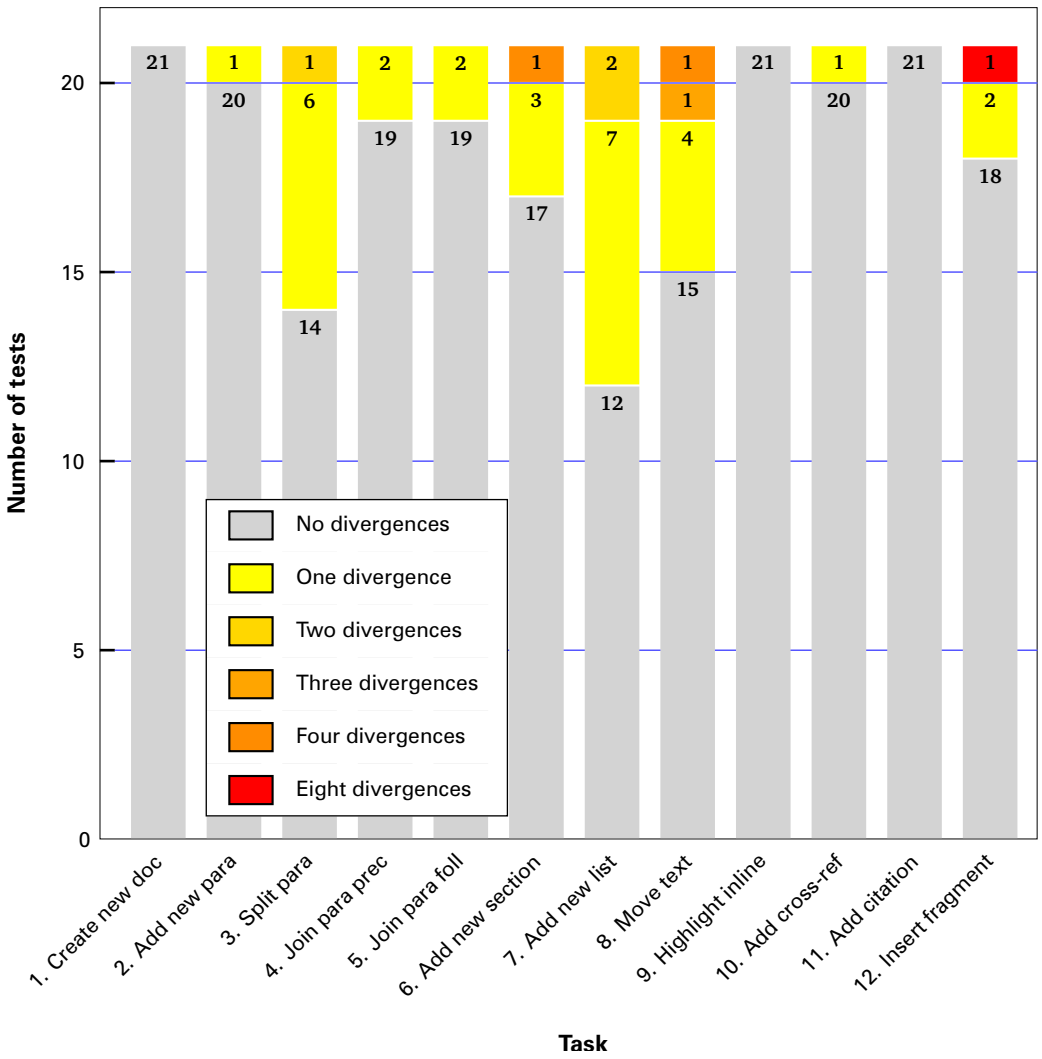


Figure 5.6: Number of testers and number of divergences from the expected patterns

With a small data set the scope for sophisticated statistical analysis is limited. There are two principal axes of measurement:

1. how much use the testers made of the new functions in the interface; this is measured by the rank value and the affordance class of the pattern eventually followed in each task;
2. how much they diverged from the expected patterns; this is measured by

the number of unexpected steps along the way to completing a task.

Using the pattern number as a form of rank for the purposes of display, as in Figure 5.7, reflects the nature of the affordances which we need for analysis. The amount of different patterns that were used for a task reflects the greater diversity of methods perceived by the tester, or 'known' from prior experience. In this approach, tests 10–12 would be ranked the most successful, as all testers but one ended up using a single pattern, although we can seen the divergences in Figure 5.6 on the previous page. By the same measure, tests 6 and 7 would rank the most diverse because of the different ways of solving it, but in these tests there were several different ways to achieve the result using the new affordances.

This approach therefore has some disadvantages when looking at aggregated results, despite its finer gradation. It implies a discrete stepwise differential

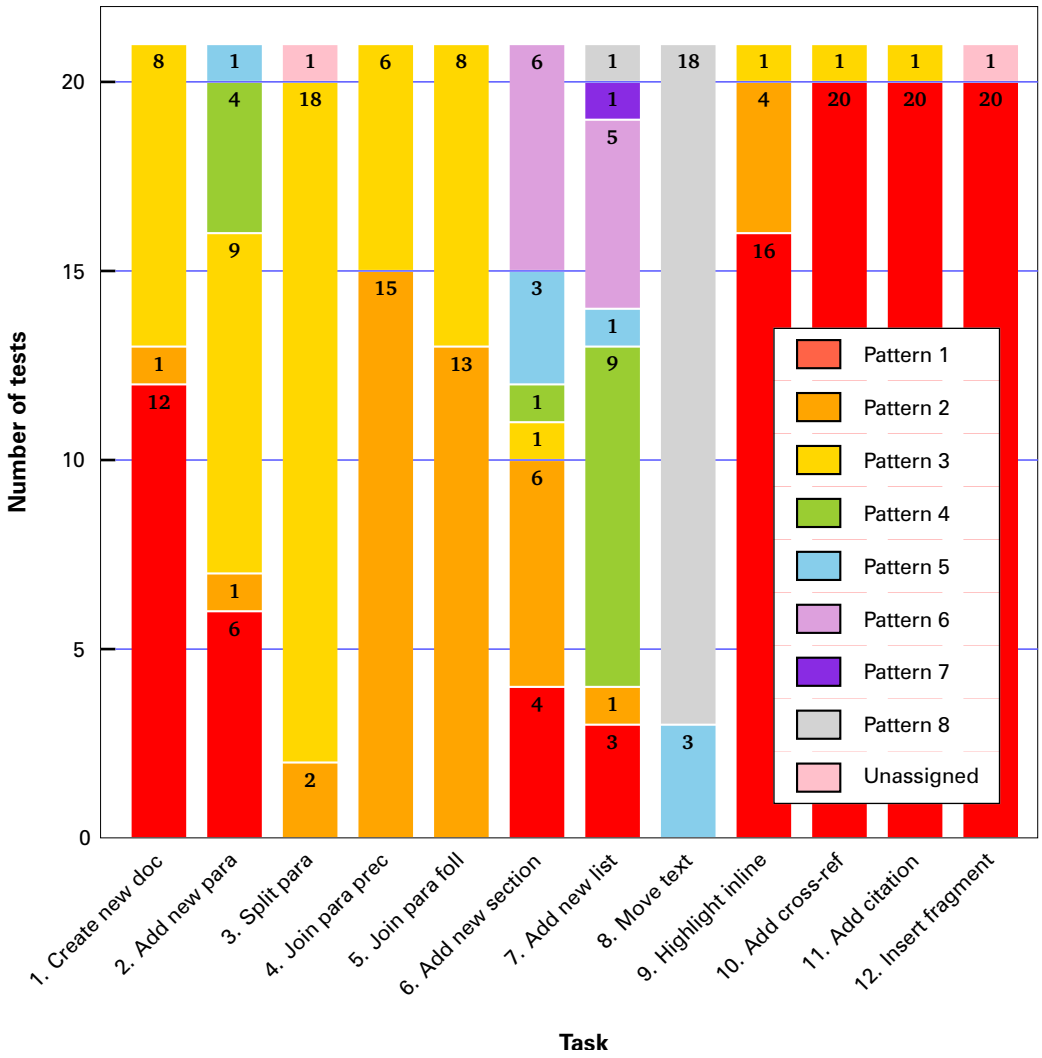


Figure 5.7: Number of testers by task showing pattern eventually used

between patterns, whereas it could be argued that the nature of the patterns is actually a continuum, and should be measured with a real value rather than an index. However, this would introduce an element of personal judgement (in choosing the values that would distance one solution from another), and with a small data set it is important to avoid the potential for this kind of bias. A further difficulty is that there is not the same number of patterns for each task, so there is little comparability between them. A final problem is that the data is discontinuous (not every test ended up using all the patterns, but only those the testers actually used).

We can also therefore look at the 'affordance class' variable which we had assigned to each pattern in Figure 5.2 on page 299, where a value of `n` indicates 'new', `o` means 'old', and `on` indicates a hybrid of old and new. This can be used to see whether or not the pattern uses *any* of the new affordances or not, and is in effect a compression of the pattern rank into categories of use.

Using this to segment the results for each task into old, new, and mixed, it is clear that there are three pairs of *related* tasks for which the affordance class of the solution was very similar, if not identical (shown bracketed under the x-axis in Figure 5.8 on the following page in the PDF output but not in the EPUB):

1. joining paragraphs (4 & 5), all done using existing affordances;
2. adding a new section or a list (6 & 7), done with approximately half new affordances and half mixed;
3. adding cross-references and citations (10 & 11), done almost entirely with new affordances.

Two apparently unrelated tasks (1 and 12) also share a similar response (curiously both are connected with file-management tasks: creating a new document, and inserting a fragment from a document). The other four tasks vary more widely in the mix of affordances used, and they are in any events discrete and unrelated in their nature.

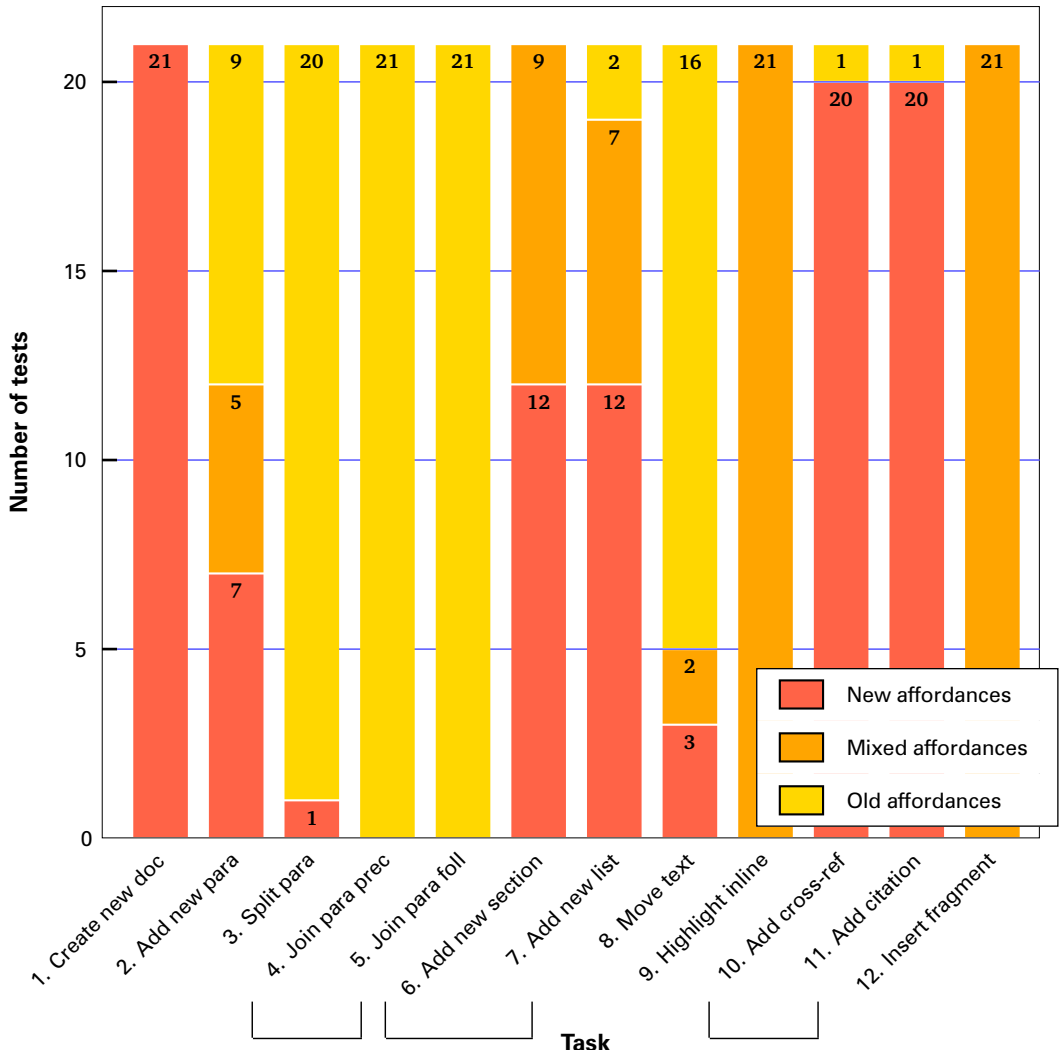


Figure 5.8: Number of testers by task showing affordance class

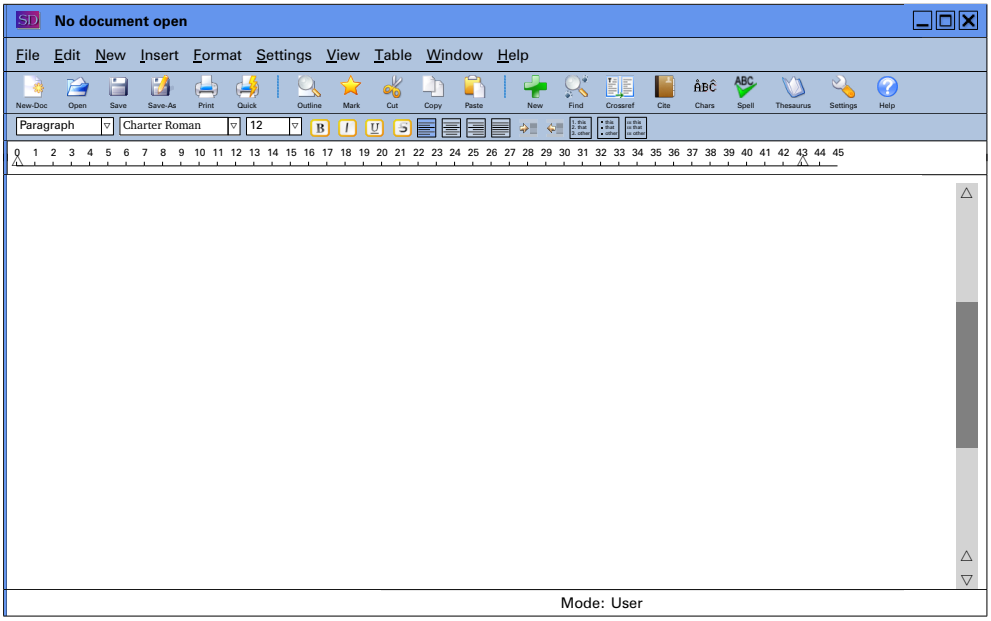
5.2 Individual test results

The test protocol included an explanation of all the new affordances we introduced (page 2 of the script in appendix section D.1 on p. 404). Each one was identified by pointing to it while the rubric was read, and an indication of assent was gained from the tester before moving to the next one.

A few of the testers paused in some tests, 'hunting' with their index finger before selecting the menu or toolbar button that they finally chose. These pauses were infrequent, and not recorded, so it is not possible to associate them with any of the divergences.

The following 12 sections show the initial screen shown to the testers, and a block (area) chart showing the number of steps to completion taken for the various patterns assigned: the height of each block represents the number of testers who ended up using that pattern, and the width of the block represents the number of steps involved. An analysis of the variant and divergent keystrokes and mouse-clicks is in section 5.3 on page 329.

5.2.1 Test 1: Create a new Article Journal document



This was the only task completed entirely using the new affordances:

- over half the testers (12) used the New-Doc button;
- one used the New button;

- the rest (8) used the New menu.

This meant that the 12 who used the New-Doc button completed the task in two steps; the remainder used three steps (see Figure 5.9).

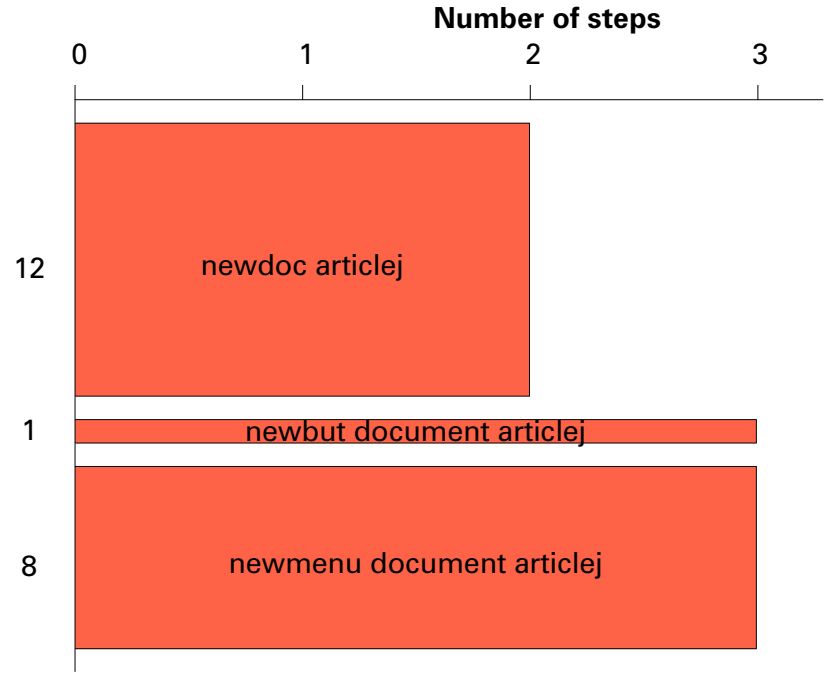


Figure 5.9: Completion of Test 1: new journal article

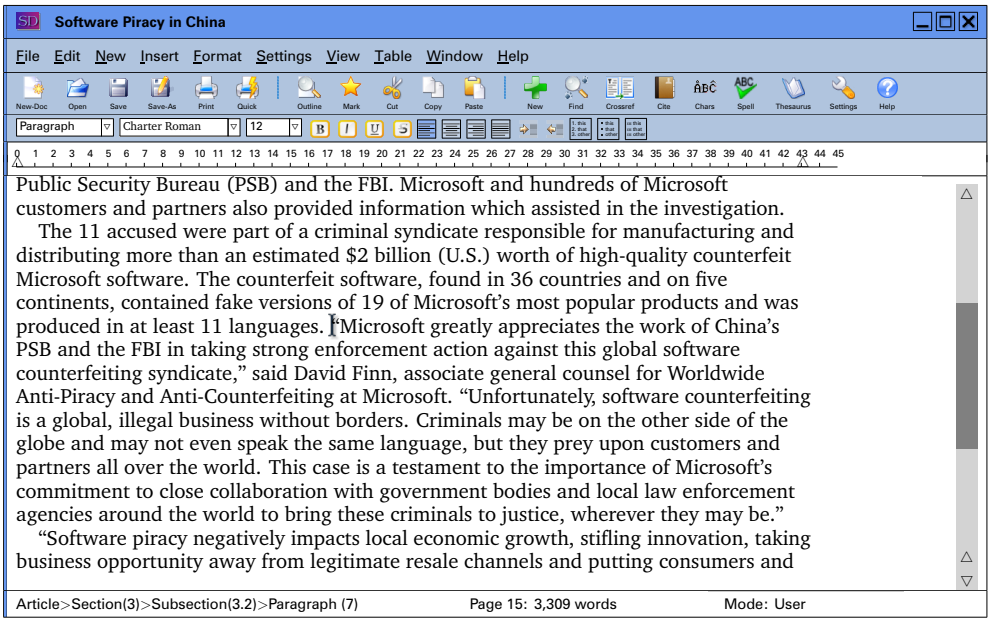
The convention in wordprocessors has been that there is no such thing as a 'type of document' (except possibly if a formal stylesheet or template is being used, which is rare): although recent versions do now provide some variety of pre-designed layouts, they do not implement named styles. In structured editors it is essential, otherwise there can be no control or consistency in the formation of the document. For this reason it is standard practice for structured editors to supply a set of popular DTDs or schemas with matching stylesheets, and to provide a facility for corporate document controllers to add their own.

This means that the concept of selecting a document type before starting to type is largely foreign to many users. Providing an affordance to perform this task appears to be an acceptable method, although it is largely a graphical equivalent for the File | New menu.

However, testers were explicitly asked to 'create a *new* Journal Article'; this was the first test, so the explanation of the new affordances from the rubric would have been very recent; and the starting screen for this test was the base window with no text. These factors combined may have provided sufficient impetus for

them to seek out the New Document function.

5.2.2 Test 2: Add a new paragraph after the current one



This is the first of three tests covering the addition of new material, for which the

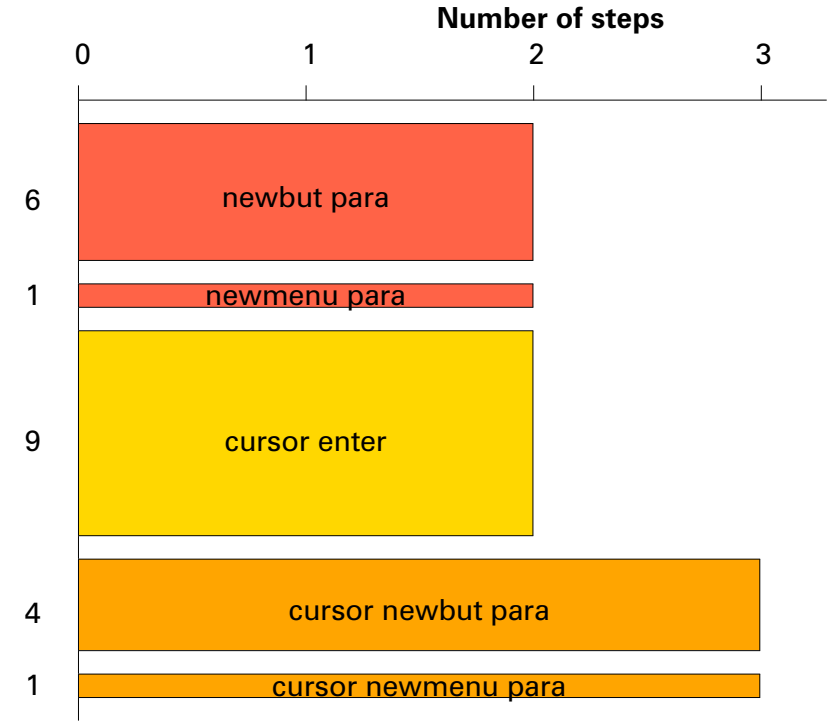


Figure 5.10: Completion of Test 2: add new paragraph

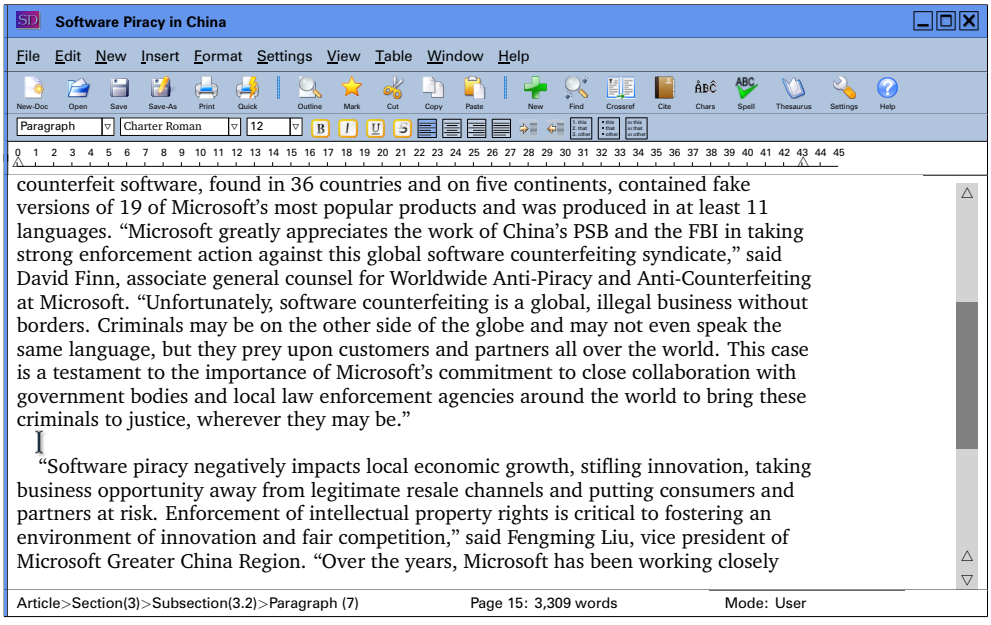
New function (button or menu) was provided and explained in the rubric. The cursor is positioned in mid-paragraph, to test the concept that adding a new element would automatically do so in the next available location (in this case, after the end of the current paragraph). This is a wholly new feature, but it was not specially emphasised in any way apart from being mentioned.

Two-thirds of the testers (14) ignored this, and their first move was to position the cursor correctly at the end of the paragraph. However, one third used the New function as their first move.

Of the two-thirds who moved the cursor, five then used the New function. The remainder pressed the Enter key, which is the traditional method.

As either of the two-step solutions achieves the identical result, it is arguable that there is no gain in efficiency in the new method, but in the case where the end of the paragraph (or whatever the current element might be) is well below the bottom of the screen, there would be time saved in not having to scroll down, or search visually to find the end-point.

5.2.3 Test 3: Split a paragraph into two



The cursor in this case is where it was left at the end of the preceding test, so it was made clear in the rubric that the paragraph to be split was the one the tester had just left, above where the cursor is.

The word 'split' caused some difficulty and had to be explained to half the testers

(10). During the explanation, it became clear that they thought it meant 'add a new empty paragraph in the middle of the existing one', and this was the cause of the non-match for one tester. Perhaps a more explanatory wording would have been clearer, such as 'make this paragraph into two paragraphs'. There was also surprise among a few testers that the paragraphs were formatted with indentation and no space between them, and it had to be pointed out that this was a WYSIWYG display showing how the document would appear when published.

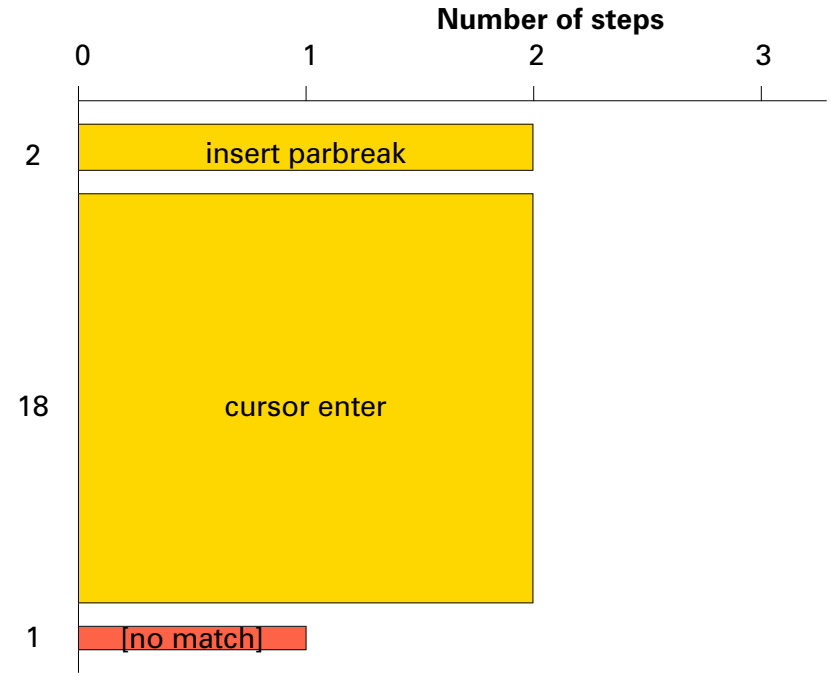
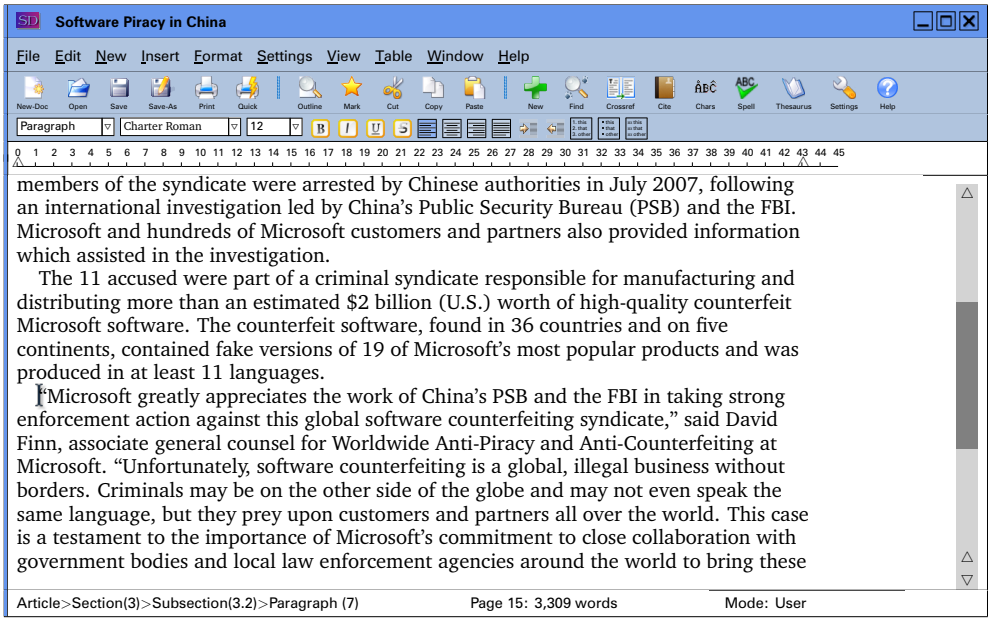


Figure 5.11: Completion of Test 3: split a paragraph

The majority then went immediately for positioning the cursor and pressing the Enter key, which is the standard method in most systems. The two using the Insert function were both experienced academic users, possibly more accustomed to the detail of editing other people's text for publication.

As we saw in the first item on page 144, the Split function is standard in almost all structured editors, and the Edit menu contained entries for both joining and splitting, but was used only once. While it may not appear to differ from using the Enter key, its use when markup is displayed means the cursor could (for example) be resting *outside* a paragraph — and in that case, 'split' would mean splitting the enclosing (parent) element, perhaps a list item or a table cell. That would be a much more radical step than splitting a paragraph or a list item, and one requiring a program to check with the DTD or schema to see if the document structure would allow it.

5.2.4 Test 4: Join a paragraph to the preceding one



The reverse of the Split function is Join. Rather than ask the testers to re-join the paragraph they just split, this test applied to the part of the paragraph *before* the split, which they had to join to the preceding one (in the screenshot above, joining '. . . the investigation.' to 'The 11 accused. . . '

This was universally solved by positioning the cursor and using the Delete or

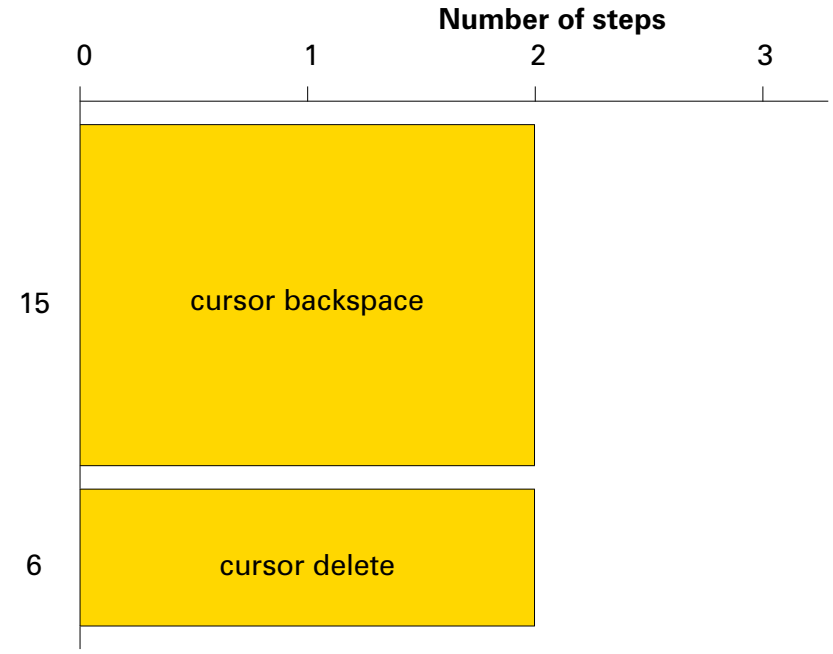


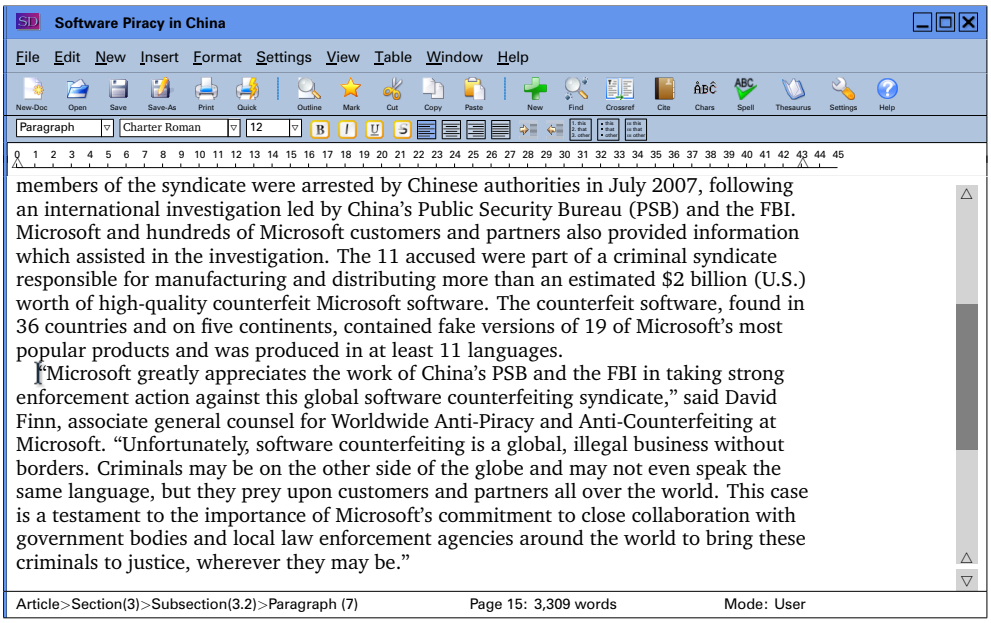
Figure 5.12: Completion of Test 4: join paragraph to preceding

Backspace keys. Although the Edit menu contained the Join function, and attention was drawn to it in the initial rubric, it was not used at all.

Three-quarters of the testers positioned their cursor at the start of the paragraph and used the Backspace key to join it backwards to the preceding paragraph. However, five testers positioned their cursor at the end of the preceding paragraph and used the Delete key.

There appears to be a slight preference to make the deletion of the paragraph break in the direction *from which* the user moves the cursor — the cursor in this case being positioned below the breakpoint, and most testers used Backspace. This is supported by the results of the next test, where the cursor was positioned *above* the breakpoint, and the Delete key was used more. However, the sample is too small and the options too restricted for this observation to be generalised.

5.2.5 Test 5: Join a paragraph to the following one



This test is the analogue of the preceding one, where the paragraph must be joined to the one below it, which in this case is deliberately off-screen (this was explained to the testers to reassure them that there was indeed a paragraph down there to be joined to).

Again, the response was to use the cursor movement and the Delete or Backspace keys. No attempt was made to check the Edit menu. The divide between Delete-users and Backspace-users was not as marked here as in the preceding test,

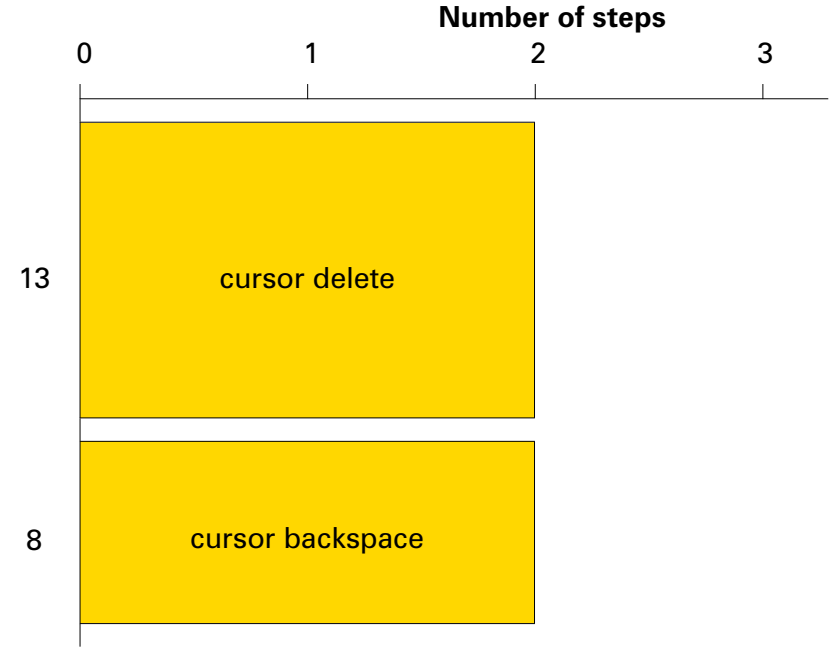


Figure 5.13: Completion of Test 5: join paragraph to following

being close to two-thirds for Delete (forward/downward movement) and one-third for Backspace (backward/upward movement).

In the discussions which followed the tests, testers were asked if they had a preference for using the Delete of Backspace key for removing characters and white-space. There was no marked preference, except that five (older) users expressed a preference for the Delete key regardless of the direction in which they had to move to reach the point where they wanted to act. When pressed, three of them said they were uncertain that a Backspace key would erase leftward or upward. They were all confident that Delete would erase rightward or downward.

It is possible that this is a residual memory of early experiences with computers (given the age of these testers), where the Backspace key did indeed sometimes come configured as a non-destructive back-movement, similar to the left-arrow, which may be a legacy of early Unix systems operated over half-duplex serial connections. However, it is not supported by the evidence, at least not in the previous test, where the majority used the Backspace key. Perhaps more likely is the idea that 'joining-to-previous' is an inherently backwards-moving action when approaching the position from lower down the document, and 'joining-to-following' is an inherently forwards-moving action when approached from earlier in the document. This is tested in the panel 'Is there a preference for Backspace vs Delete according to the direction of joining paragraphs?' on the next

page and may support this hypothesis.

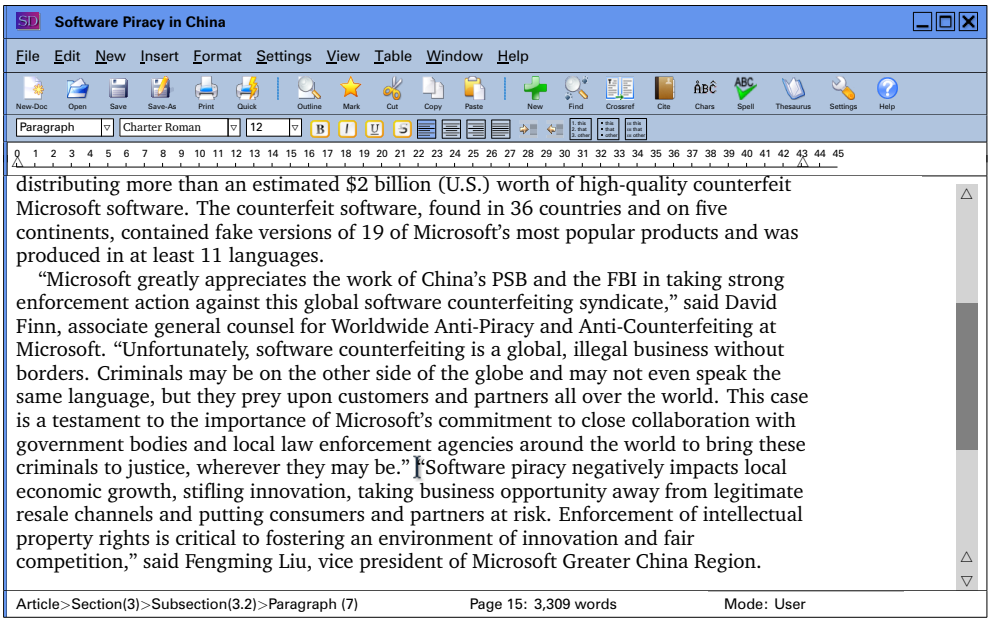
Unusually in these data, the rectangular 2×2 design of Tests 4 and 5 allow us to make a measurement of the probability of the results not being associated. With a 2×2 design, a χ^2 test is inappropriate — given the degrees of freedom of 1 and some of the data being below 10, the expected frequencies would be small, and the χ^2 distribution is conservative with probabilities for small values. Instead, we computed Fisher's Exact Test:

	4. Join to Preceding	5. Join to Following	Total
Backspace	15	8	23
Delete	6	13	19
Total	21	21	42

This gives us $p = \frac{23!\times19!\times21!\times21!}{15!\times8!\times6!\times13!\times42!} \approx 0.025$ (using *KCalc*, a common desktop application). Such a low value would lead us to reject the null hypothesis that the choice of using Backspace or Delete is unconnected to the nature of the tasks. This is in line with the (untested) hypothesis that there is a directional preference.

Panel 5.1: Is there a preference for Backspace *vs* Delete according to the direction of joining paragraphs?

5.2.6 Test 6: Add a new section to the article



This test and the next one both involve adding new structural material to the document. As with Test 2 (Add Paragraph), the facility exists with the New menu (and button) to add the element without having to position the cursor first, as explained in the rubric.

All the testers used the New button or menu, but just under half (9) positioned the cursor to the end of the current section manually first. Two used the Outline function to navigate to the end of the section.

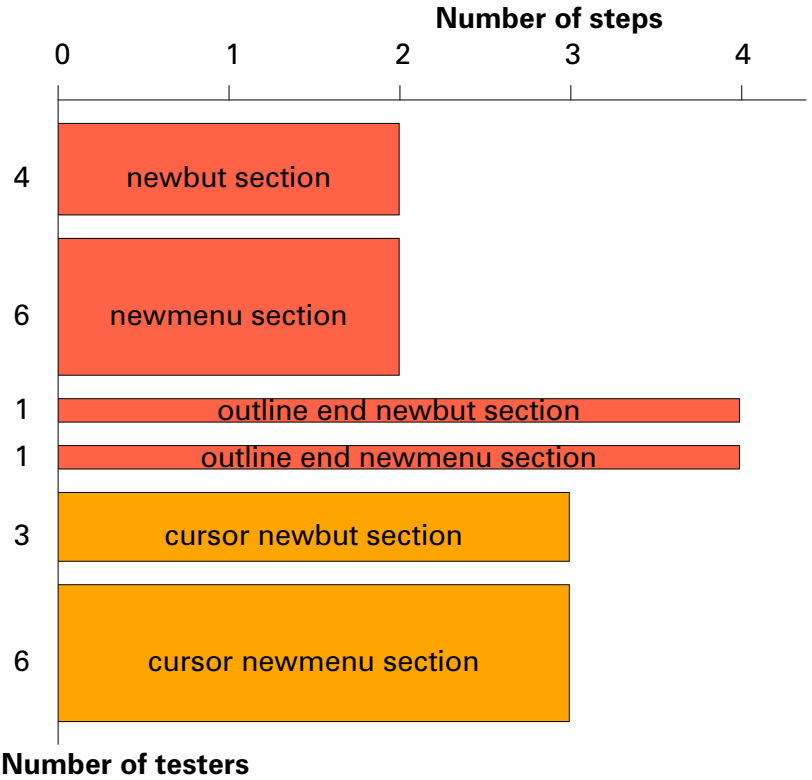
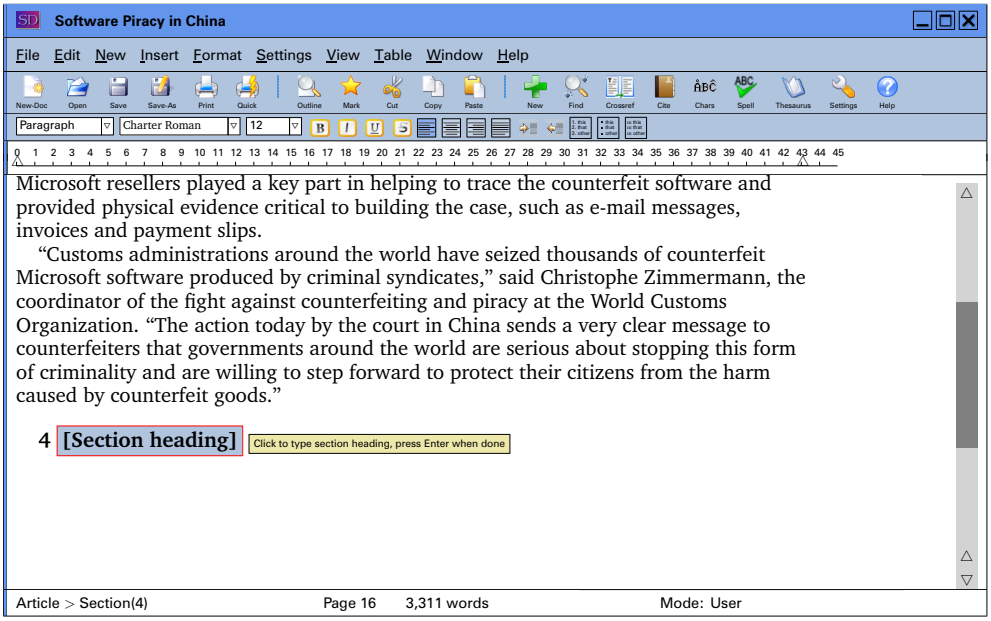


Figure 5.14: Completion of Test 6: add new section



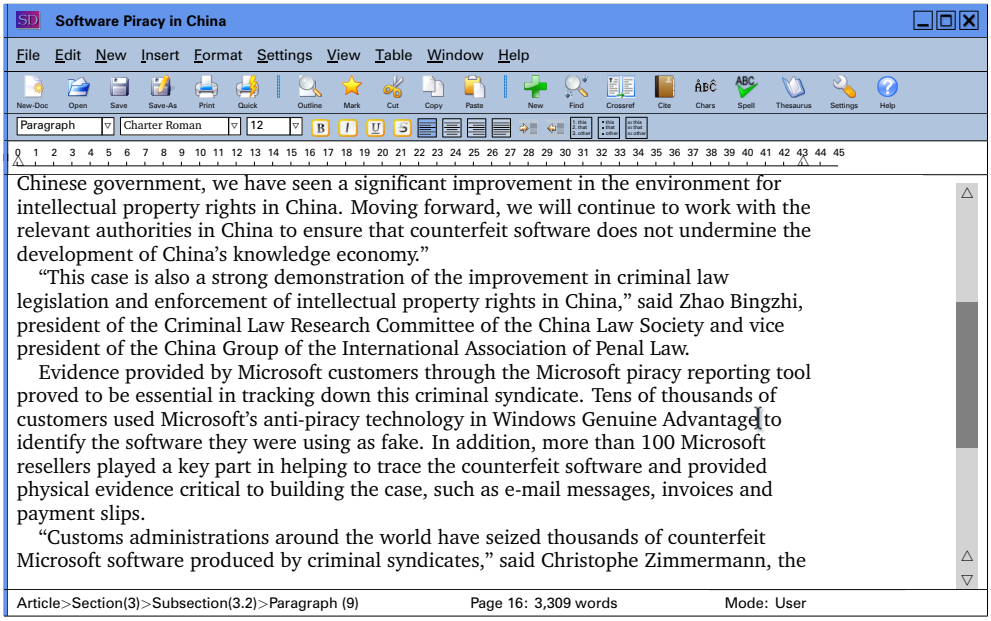
Surprise was expressed by most testers that the result was a fully-formed section heading, complete with number. The concept of an editor which 'understands' document structure was clearly alien to most of testers, even those with a familiarity with *Word*'s named styles.

5.2.7 Test 7: Adding a new list

This test and Test 9 involve the assignment of a variant behaviour to an existing affordance. This is a change which is normally deprecated: changing a long-established behaviour breaks the user's mental model of the activity, and can be disruptive when other, possibly competing, applications retain the original behaviour.

However, in these cases — as with the introduction of the New button and menu — the change is minor, and the benefit significant.

The change envisaged for this test is that the List buttons on the toolbar would *no longer make the current paragraph into a list item*, but would insert a new list in the next available location (here, immediately after the end of the current paragraph). In practice, the existing behaviour of toggling list-item format on and off could be retained in cases where one or more paragraphs were highlighted.



Notwithstanding this change, 12 of the testers used the numbered-list toolbar button: nine moved the cursor first and three did not. With one exception, the remainder used the New button or menu, most with a positioning movement first. Only one tester used the traditional method of starting a new paragraph by typing a digit 1 followed by a full point and a space, which is automatically

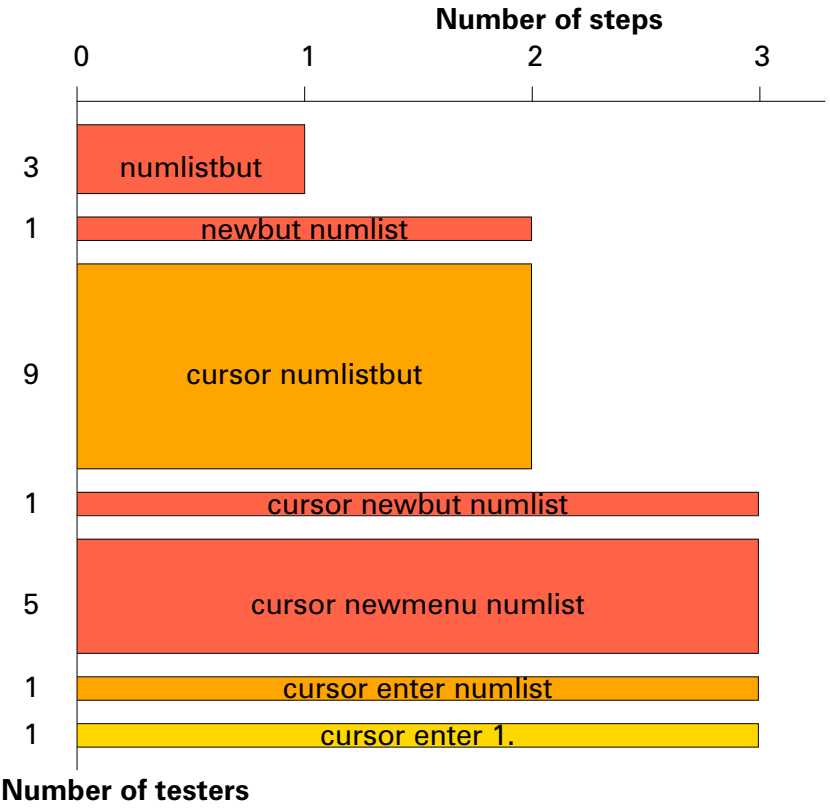
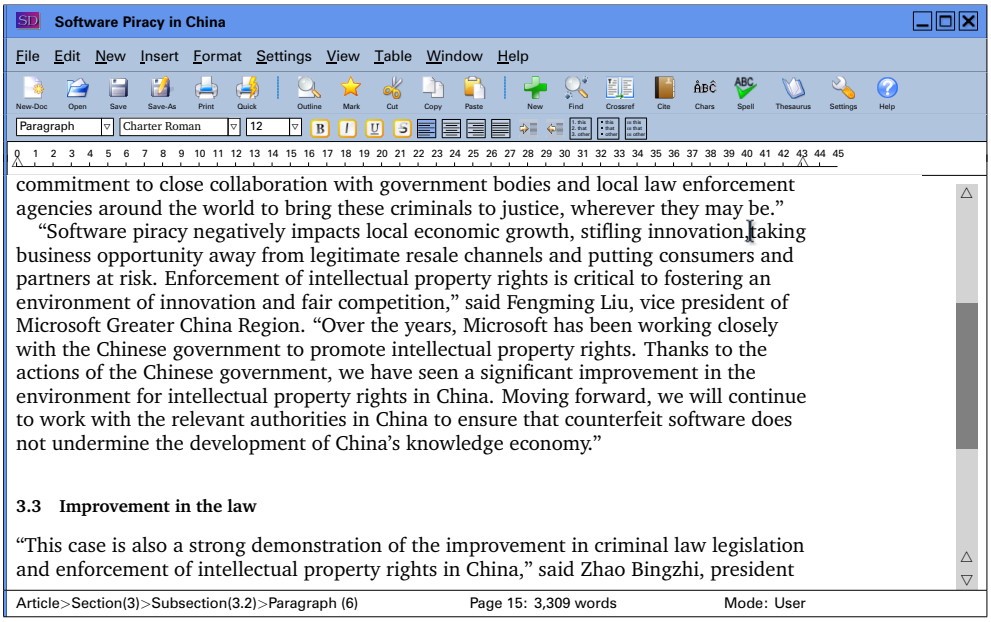


Figure 5.15: Completion of Test 7: add new list

detected by most wordprocessors as the indication that a new list should start.

In post-test discussions, the variant behaviour was discussed with some testers, who understood that it was different but felt that if the existing behaviour was retained for highlighted paragraphs, it would not be disruptive, as the Toggle List function was anyway typically only used on a group of highlighted paragraphs.

5.2.8 Test 8: Move a block of text from one place to another



There are typically two pieces of excise involved in moving a block of text. One is the manipulative effort in highlighting it, especially if it extends to more than a single object, or reaches down below the bottom of the window. The other is the effort involved in scrolling or otherwise moving the cursor to the destination. In the case of a very long document, perhaps many hundreds of pages, leaning on the down-arrow button or the mouse-button on the scrollbar can be both unproductive and tedious.

An outliner, such as described in section 2.2.1.4 on page 81 lets the writer locate the destination, or at least its containing structural element (a subsection, for example), and then use the cursor keys or mouse briefly to locate the exact position.

Here, in a shorter document, the second excise point is not onerous, and can be ignored. For the first, despite the explanation in the rubric of the Mark button for highlighting a whole object, only three testers used it, one of whom also

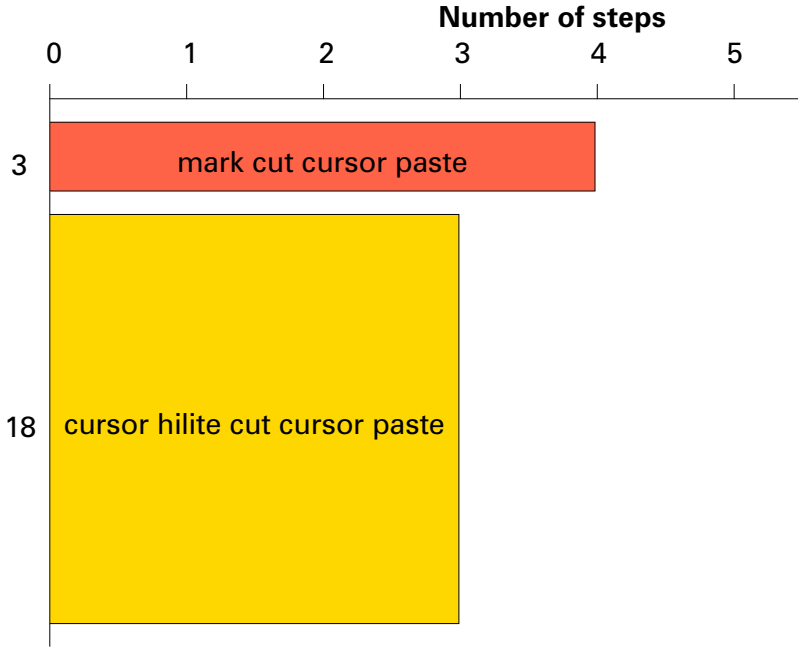


Figure 5.16: Completion of Test 8: move text block

exercised the Ctrl-Shift-Uparrow combination (a *vi* function).

The remainder (18) all highlighted the text with the cursor, cut it, moved the cursor down to the target location (by scrolling or using the slidebar; one used Find) and then pasted the text.

Only three of these pressed the Enter key to 'create the new paragraph' before pasting, which raises a fundamental point about users' expectations when cutting *all* the text from an element like a paragraph: should the program also remove the now-empty paragraph markers as well, or leave them behind? In effect, is the writer cutting the entire paragraph, or merely its content?

If it is the entire paragraph, then when it disappears at a Cut operation, the surrounding elements close up the gap, and when it is pasted, it needs no blank-line preparation, as it is a wholly self-contained object which the program would know how to deal with.

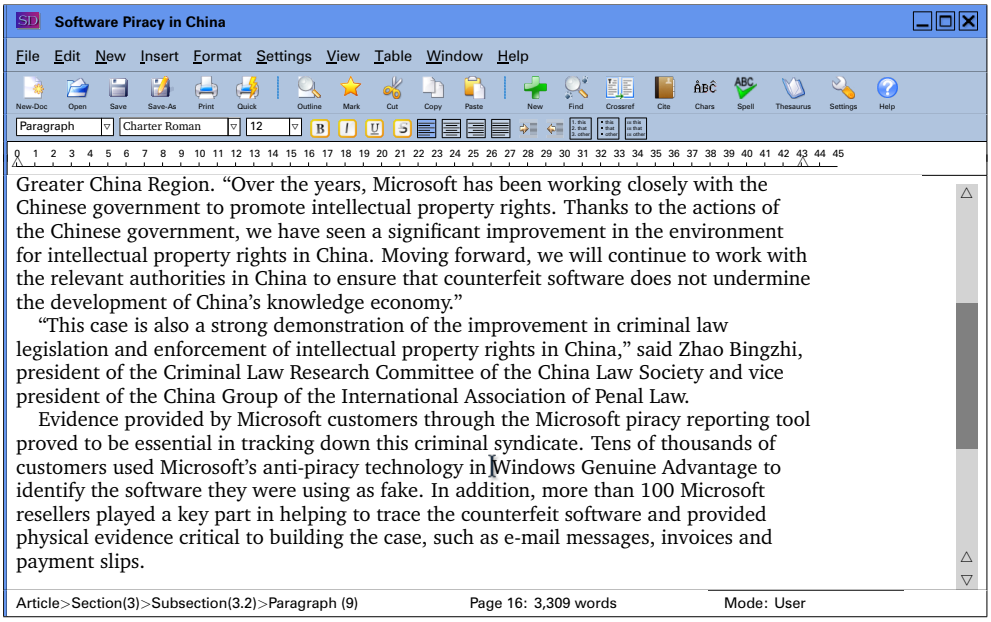
If, on the other hand, only the content was removed, leaving behind an empty paragraph, then pasting at the end of another element would simply insert the text into that element: to paste it as a paragraph in its own right, an empty paragraph would need to be created first.

In reality, an editor can be programmed to notice when all the content has been removed from an element, and to delete the empty element as a matter of course.

It would then 'know' that the material was originally a whole paragraph, and behave accordingly when it is pasted. However, if the writer actually wanted to take the whole content of an element and paste it *into the middle* of an existing block of text, the program would need to understand that this would mean ignoring the paragraphic nature of the cut text, and to merge it with any surrounding text when pasted. This would be in accordance with the principles of Smart Insertion, which we have mentioned elsewhere.

This test also revealed a minor omission from the screenshots. To accompany the Edit | Move menu entry there should have been a [Move] button, as for the other new affordances. However, no tester appeared to look for it (not even in the Edit menu, where it was present).

5.2.9 Test 9: Highlighting a trade name (product name)



This test concerned the second variant behaviour on an existing affordance, adding a drop-down menu to the font-change buttons [B], [I], and [U]. Only italics were tested here. The objective is to capture the semantics of the font change, which is always more important than the actual italicisation of the letters, especially for retrieval or subsequent reprocessing or repurposing..

The testers were required to highlight a product name in italics (the rubric was carefully chosen to emphasise the reason for the change [a product name] *before* the visual appearance [italics]).

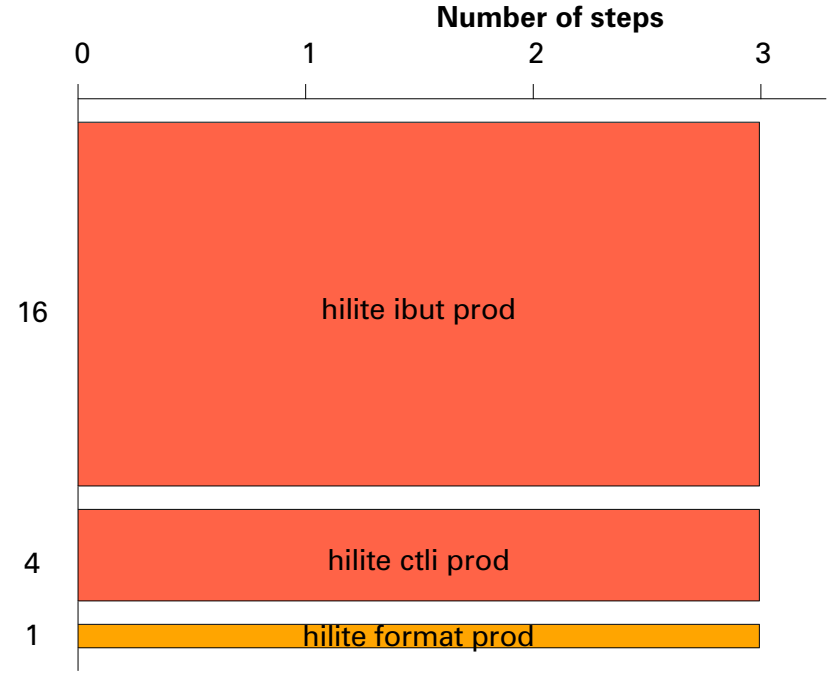
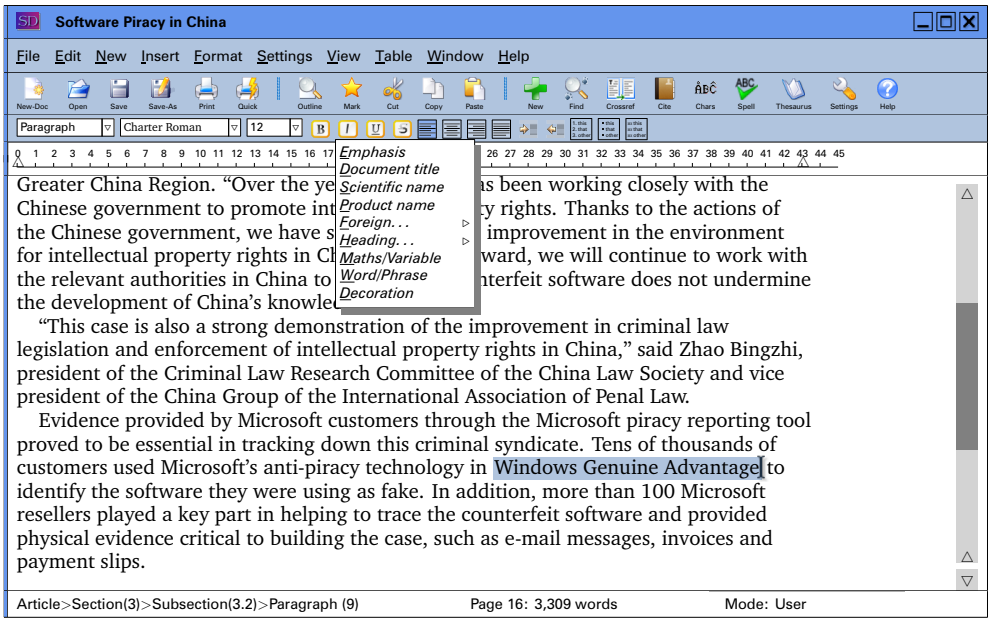


Figure 5.17: Completion of Test 9: highlight product name

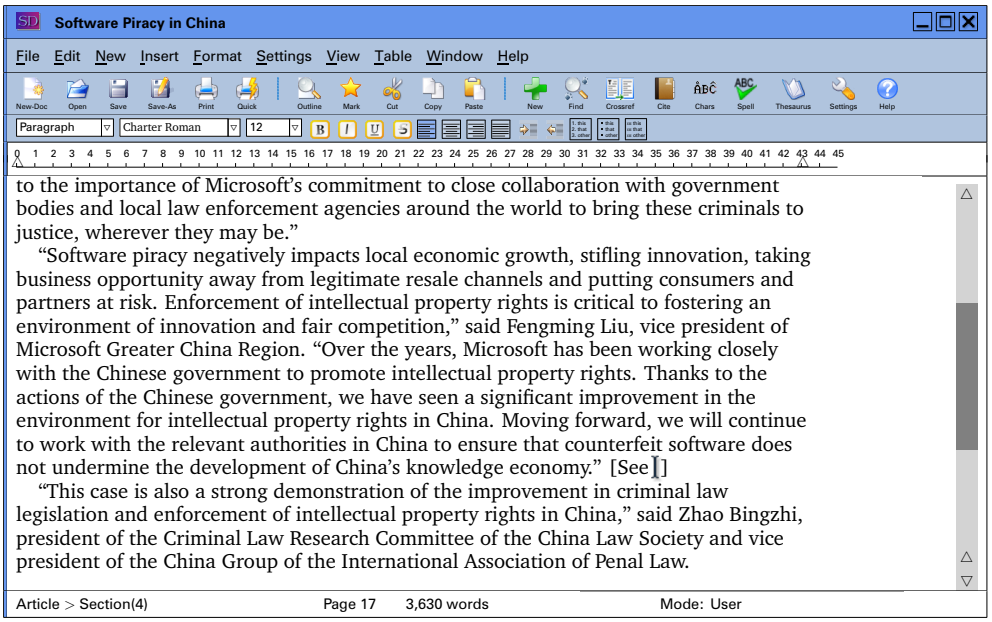
All testers bar one used the I button or the Ctrl-I key combination; the other tester used the Format menu, which amounted to the same thing. Some initial surprise was evident at the appearance of a menu, but this was extremely brief as testers saw the Product Name entry.



This is the only test which has been replicated independently so far, and the result accords exactly with the those recorded in the study by Geers (2010, §§ 5.2, 6.4.5) which we referred to earlier in section 4.3.4 on page 253.

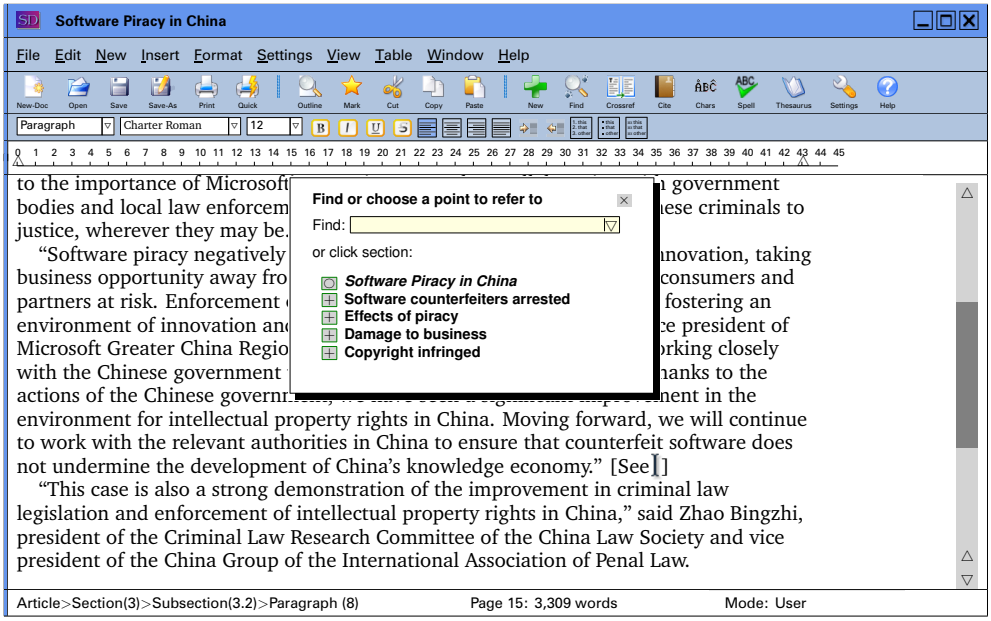
Some users thought that the question was about repeat-replace editing, but they asked first, and it was explained that only the single indicated instance of the product name was to be highlighted.

5.2.10 Test 10: Add a cross-reference to another section



Making an 'automated' cross-reference traditionally involves first setting a marker or label at the target location and then returning to the point of reference to insert the link. The new affordance pops up a list of referenceable locations (in the example, a list of section and subsection titles), and a single click inserts the cross-reference. The Cross-Ref button was used by all testers except one, who used the alternative Insert menu entry instead.

Recent versions of *Word* and some other systems now implement this method, based on the use of certain named styles such as `Heading~1`, `Heading~2`, etc.



However, creating a reference to a location that is not already 'special' in some way would still mean visiting that location and inserting the marker or label, and then returning to the point of reference. Alternative ways of managing this process were explained in section 4.3.6.2 on page 271 but were not tested, as the instantiation of sliding and fading lightbox windows is outside the expressiveness of paper prototyping.

In the author's experience, many users forego even the 'semi-automatic' method

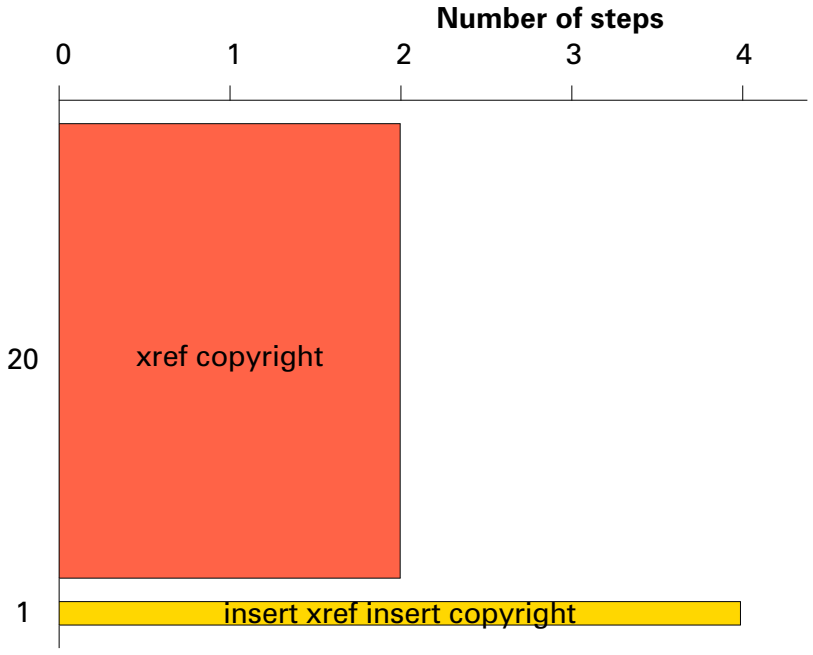
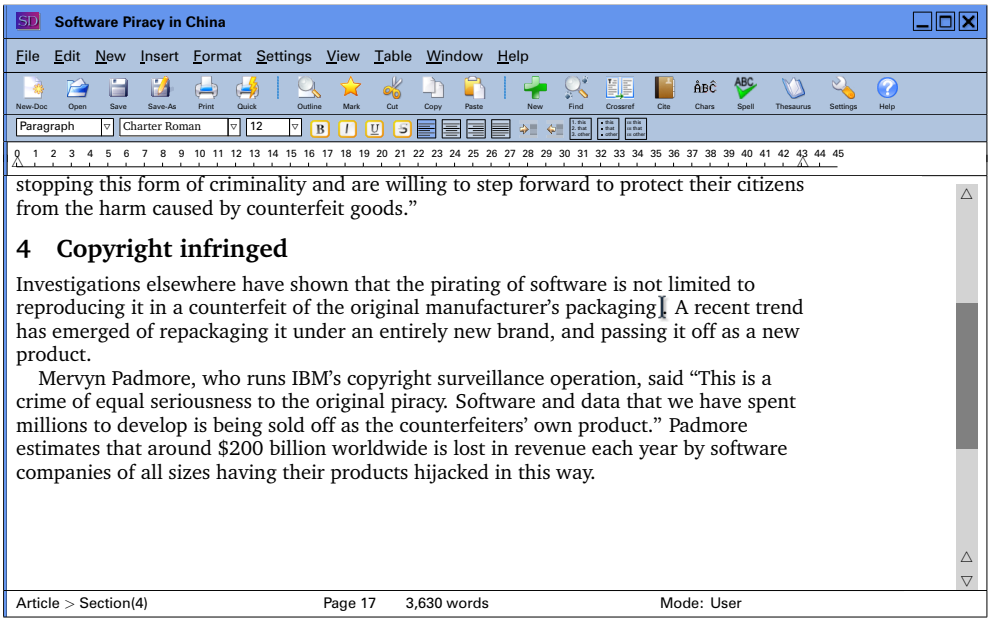


Figure 5.18: Completion of Test 10: add cross-reference

for the longer way of typing the reference by hand, even with the necessity to update it any time the document structure might cause the value to change.

5.2.11 Test 11: Give a citation to an article you need to refer to



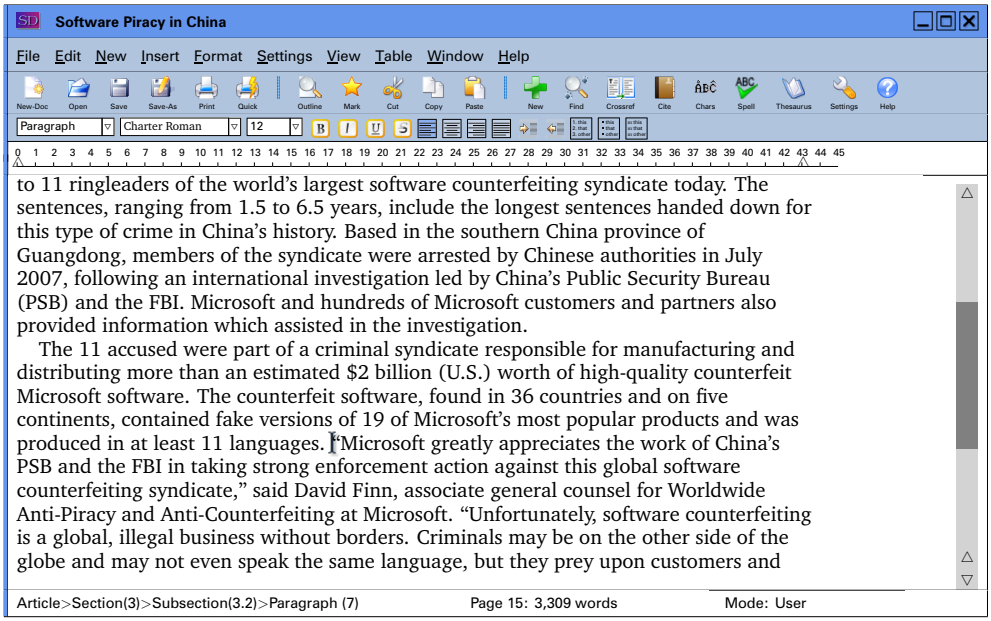
The use of a bibliographic database like *Zotero*, *Mendeley*, *EndNote* or similar, together with a suitable plugin for the editor, should make this a straightforward operation.

The principal difference in the method implied here is that the bibliographic database would not need to be opened as a separate application (and kept open during writing), but would pop up only when the affordance was selected. This was commented on by several testers as contributing to a reduction in desktop clutter.

This test implements such an affordance as a Cite button and as an entry in the Insert menu. All 21 testers ended up using these.

In this author's experience, however, many writers still type their citations and references manually, or copy and paste them from another document, reformatting them to match the publisher's style by hand. Indeed, three of the testers volunteered the information that this was their current method of working, but having seen how an automated method might work, were now moved to try it.

5.2.12 Test 12: Insert a fragment of another document



Copy and paste of structural elements from another document is a commonplace occurrence, although tedious and error-prone. Multiple windows, different formatting, and the problems of mouse control over selection and pasting contribute to slippage and inaccuracy.

Using markup could overcome some of these problems, especially when the two documents share a common structural vocabulary, which is what we are testing

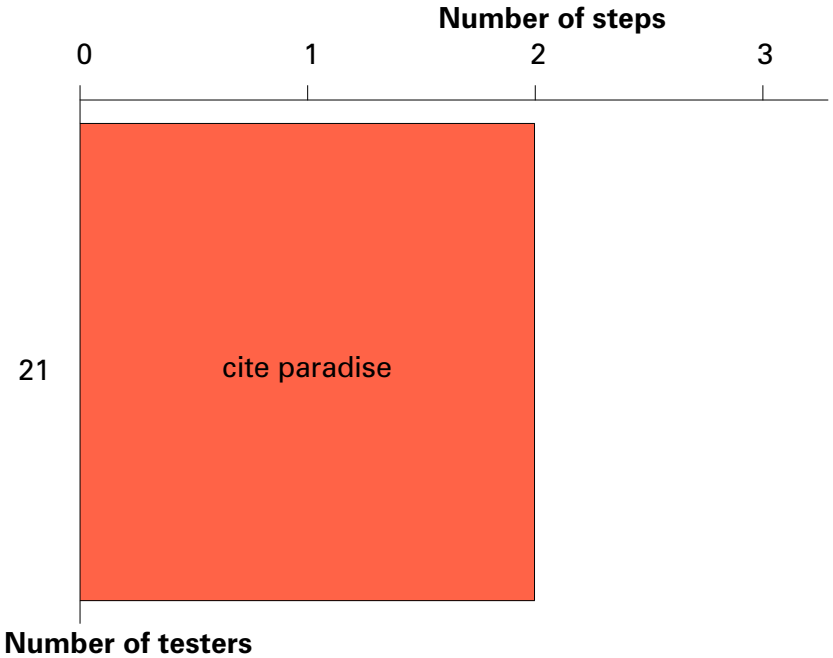
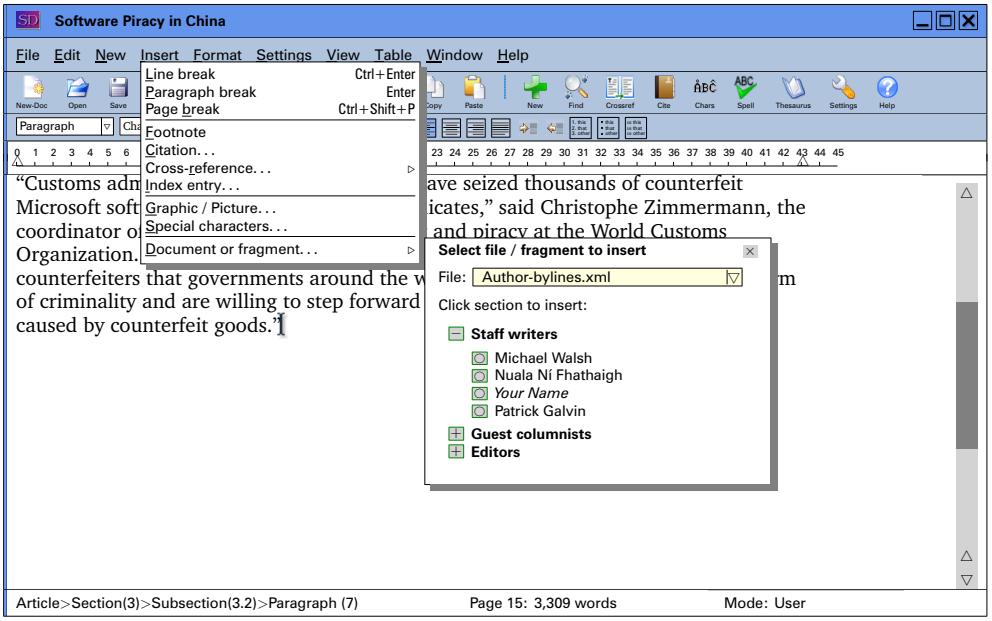


Figure 5.19: Completion of Test 11: add citation

here. The task is to insert a fragment (a 'by-line' signature block) from another document into the end of the current document.

As elements such as by-lines are identifiable as such in the structure, and as the location and formatting is known to the stylesheet, it is not necessary to open the document as if for editing and search for the by-line. A special dialog which can identify the structural elements in a selected file is all that is needed.



The File | Open dialog is elided here, as we are not testing navigational basics

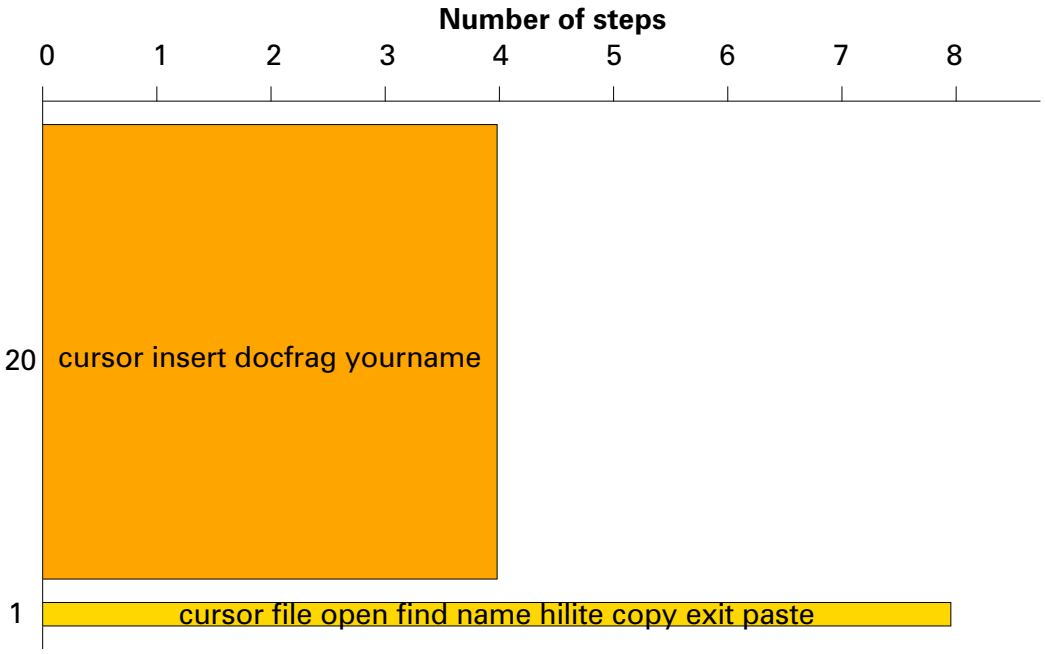


Figure 5.20: Completion of Task 12: add document fragment

which are common to all programs, so it is assumed that the relevant file is the last one used, which means it appears immediately. All but one of the testers used the Insert menu (no button was provided for this operation, but in a real interface that could be configurable). The one tester used the traditional steps of opening the file and doing the copy and paste manually.

The use of copy and paste is frequent in boilerplate editing, such as contracts, standardised documentation, or personalised documents, where a chunk of text is inserted, and then a word or two changed. Large corporate document-management systems do indeed have this kind of navigational insertion framework, but there would appear to be no barrier to it in a normal structured editor.

The use of documents from another vocabulary would require mappings to be maintained, or the use of an intervening software layer implementing the techniques of Smart Insertion and Target Markup Adoption.

5.3 Divergences from patterns in tested behaviour

The outlier of eight divergent steps, mentioned earlier, to solve one of the tasks was a tester on Task 12 ('Insert a fragment of another document') who used the traditional steps of opening the external document, finding, marking, and copying the required text, closing the document, and pasting the text into position. This was an extreme case, and perhaps indicates that the anecdotal perception of user reluctance to adopt new methods may not be entirely correct.

A coarse measure of divergence can be taken from simple aggregation of the data presented in Figure 5.6 on page 303, by multiplying the number of divergences by the number of testers who diverged (a form of weighting) as shown by Figure 5.21.

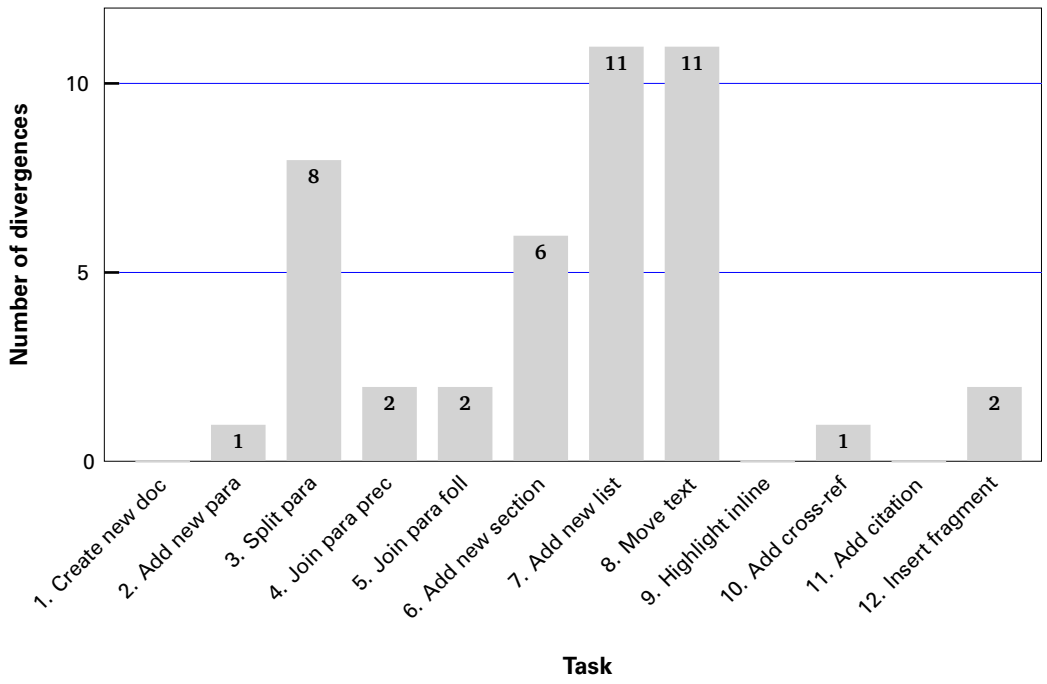


Figure 5.21: Total number of occasions of divergence by test

Three tasks (1, 9, and 11) were completed with no divergences at all, and five more (2, 4, 5, 10, and 12) with only one or two. The list of divergences is shown in Table 5.1 on the following page.

As the total number of divergences is small (35), even with a small overall sample ($21 \times 12 = 252$), no classification of them has been attempted. In 20 cases (57%) the tester did in fact achieve the task using one of the new affordances.

The tests with four or more diversions are:

Table 5.1: Individual divergences (in italics)

Task	Tester	Divergences	Keystrokes/Mouseclicks										Affordance	Pattern eventually matched
2	8	1	cursor	enter	*enter*								o	cursor enter
3	5	1	*cursor*	insert	parbreak								n	insert parbreak
3	8	1	cursor	enter	*enter*								o	cursor enter
3	9	1	cursor	*newmenu*	enter								o	cursor enter
3	14	1	cursor	*newmenu*	enter								o	cursor enter
3	15	1	cursor	*newbut*	enter								o	cursor enter
3	21	1	*cursor*	insert	parbreak								n	insert parbreak
3	1	2	*newmenu*	*para*									–	—
4	8	1	cursor	backspace	*backspace*								o	cursor backspace
4	16	1	*ctlz*	cursor	backspace								o	cursor backspace
5	8	1	cursor	delete	*delete*								o	cursor delete
5	19	1	cursor	*cursor*	backspace								o	cursor backspace
6	3	1	cursor	*enter*	newmenu	section							on	cursor newmenu section
6	7	1	cursor	*enter*	newmenu	section							on	cursor newmenu section
6	20	1	outline	end	*enter*	newmenu	section						n	outline end newmenu section
6	17	3	*insert*	*linebreak*	*linebreak*	newbut	section						n	newbut section
7	1	1	*newbut*	numlistbut									n	numlistbut
7	4	1	cursor	*enter*	numlistbut								on	cursor numlistbut
7	5	1	cursor	*insert*	newmenu	numlist							n	cursor newmenu numlist
7	7	1	cursor	*enter*	numlistbut								on	cursor numlistbut
7	8	1	cursor	*enter*	numlistbut								on	cursor numlistbut
7	17	1	cursor	*insert*	numlistbut								on	cursor numlistbut
7	21	1	cursor	*enter*	newmenu	numlist							n	cursor newmenu numlist
7	6	2	cursor	*format*	*insert*	newmenu	numlist						n	cursor newmenu numlist
7	19	2	cursor	*enter*	*insert*	numlistbut							on	cursor numlistbut
8	3	1	cursor	hilite	cut	cursor	*enter*	paste					o	cursor hilite cut cursor paste
8	16	1	mark	cut	*ctlshfup*	cursor	paste						n	mark cut cursor paste
8	18	1	cursor	hilite	cut	cursor	*enter*	paste					o	cursor hilite cut cursor paste
8	19	1	cursor	hilite	cut	cursor	*enter*	paste					o	cursor hilite cut cursor paste
8	4	3	mark	*hilite*	*cursor*	cut	cursor	*enter*	paste				n	mark cut cursor paste
8	12	4	cursor	hilite	cut	*find*	*text*	*end*	*paste*				o	cursor hilite cut cursor paste
10	13	1	*cite*	xref	copyright								n	xref copyright
12	7	1	cursor	*enter*	insert	docfrag	yourname						on	cursor insert docfrag yourname
12	19	1	cursor	*newmenu*	insert	docfrag	yourname						on	cursor insert docfrag yourname
12	12	8	cursor	*file*	*open*	*find*	*name*	*hilite*	*copy*	*exit*	*paste*	o	—	

3 Split paragraph; as noted earlier, some uncertainty clearly existed in the testers' mental model of a paragraph. In all these cases except the last, the first move (position the cursor) was correct, but three of them (four if we include the last) then exercised the New menu or button, believing the task to require the introduction of a new paragraph. It is not known if this is due to a fault in the wording, although as the majority of testers succeeded with no problem, this is regarded as unlikely.

6 Add new section; these are minor, as they are just the introduction of unnecessary vertical white-space before finally using the new affordances to add the new section.

7 Add new list; as with test 3 above, the initial cursor movement was unexceptional (although unnecessary), but as with test 6 above, the extra white-space reveals the inheritance of the wordprocessor model, where the list buttons operate on the current paragraph.

8 Move text; these were slightly more varied divergences: some do indeed use the same model as above (of adding white-space, in this case before pasting), but there was an experimental excursion into *vi* keystroke (shortcut) mode, and a use of the Find function to locate the end of the required section.

None of these would appear to damage the thesis that the new affordances provide an alternative means to completing the tasks, but they do perhaps reveal the change of mind-set required for efficient use of a structured editor. In the post-test discussions, several testers said they were unaware that an editing interface could 'do so much'; that is, automate so much of the task as this one implies. This is discussed further in section 5.4 on the next page.

5.4 Comments and discussions

Three questions were asked after each test session was completed:

1. How obvious was it what to click on?
2. Did you feel it needed more or fewer clicks than your current system?
3. Do you feel that the program is 'doing it right' (that is, 'as you would expect')?

The responses were summarised and are listed in Table 5.2 on the next page. There was no overt discussion of these topics during the tests themselves: the script (see appendix section D.2 on p. 408) explicitly avoided all but the information needed to indicate the task, to try and avoid the problems caused by testers changing their behaviour or response in an attempt to 'do well' for the project. This risk, present in all cases where participants know they are being observed, may be seen as a variant of the Hawthorne Effect, although with no deferred post-test measurement it is not possible to see if there was a reversion to the original behaviour.

5.4.1 Informal responses

These were not intended as formal survey questions, but more designed to elicit a conversational response, although time was limited as sessions were targeted at 45 minutes, and 30 of those were nominally assigned to the tests. They do nevertheless contribute an indication of potential user satisfaction, which is overall positive.

Obviousness: Eight testers felt the interface was 'very' obvious to use; 13 rated it 'mostly' obvious.

More or fewer clicks?: 12 testers thought such an interface would take fewer clicks than their current system (mainly *Word*); six thought it would take about the same number or 'not more'; one declined to answer; one thought it would be more, but the interface was 'good'; and one would only say it was 'acceptable'.

Does it right?: 18 felt the interface 'did it right'; one declined to answer; one said 'mostly'; and one just thought it was 'easier'.

Table 5.2: Comments from testers

Tester	Obviousness	More or fewer clicks?	'Does it right?'
1	very	fewer than *Word*	yes
2	mostly	—	yes
3	mostly	same as *Word*	—
4	very	fewer than *TEXnicCenter*	yes
5	mostly	same as *Word*	mostly; needs to distinguish between new and insert
6	mostly	fewer	yes; very smart
7	mostly; document fragment menu item should use more common term	same to fewer	yes
8	very	fewer	yes
9	very	more but good	yes
10	mostly	not more	yes; easy to use; meaning of New is confusing
11	mostly	fewer	yes
12	mostly	same but better document results	yes; interface relates to a known structure
13	very	fewer	yes
14	very	acceptable	yes; unfortunately the *Word* culture is ingrained
15	very	same	yes
16	mostly	fewer	yes; make the breadcrumb clickable
17	mostly	fewer	easier
18	very	fewer	yes
19	mostly; confused between new and insert	fewer	yes
20	mostly	fewer; results more important than clicks	yes
21	mostly; new and insert confusing	fewer	yes

As mentioned earlier, the overall impression was of surprise that there even existed ways different from *Word* of doing everyday editing and writing tasks. This effect, observed by the author on many other (unrelated) occasions, is well known anecdotally to experienced user and teachers of XML, LaTeX, and other non-Word systems. There seems to be little or no research on it as a phenomenon, although there is extensive coverage in XML and LaTeX evangelist and advocacy web sites, where it is often seen as evidence of the core thesis that the commercial non-XML, non-LaTeX document-editor vendors maintain their users in a state of ignorance as a means to achieving industrial hegemony (see Cottrell (1999), for a fairly representative example)

What is clearly true from simple observation is that the predominance of a single product has had the effect of blinding users to any effect or result that *cannot* be achieved with Word. While this is mostly harmless and irrelevant in many cases (the high percentage of documents in business and elsewhere that are 'ephemeral, transient, or inconsequential' (Flynn, 2013)), it otherwise constitutes a real and serious barrier to the adoption of formal standards for managing documents.

Even the adoption of Named Styles requires a conscious effort by the author, and a significant effort by trainers and educators. Anything more than a single click is unlikely to meet with a favourable reception if the author is in fact unaware that such a feature even exists — and this is for a feature that has been in Word for nearly two decades! It takes no great step of imagination to realise that the reintroduction of once well-known concepts like structure or identity is going to require more effort than any cost-conscious organisation is likely to approve of — or it needs to be introduced by way of changes to the interface that elicit or deduce the majority of the information by way of existing affordances.

5.4.2 Criticisms

The additional comments made in answer to the questions indicate that some aspects of the interface as tested would need revising, or that a different way of expressing the intent (the affordance) should be investigated:

- the term 'document fragment' was a poor choice, as it uses a term not commonly understood. 'Paragraph' or (in the context), 'signature block' would have been better, even though the concept being tested was actually generalisable to other element types.The choice of phrase illustrates a key problem in interface design in cases where there is a noticeable difference

between the level to which users are able to generalise compared with the level to which designers, programmers and domain experts are able to.

- the distinction between New (additional structural material: hierarchy or pool) and Insert (new material inside existing text within a paragraph: flow) was imperfectly understood. Despite the extensive misunderstandings created by the application of the term 'insert' in the concept of document trees, perhaps it is already too well-entrenched to change.
- the idea is interesting that more clicks would even be acceptable if the resulting document quality is improved or the reason for the clicks is made clear.
- the breadcrumb should indeed have been clickable, as described in section 4.3.5.3 on page 269.

CHAPTER 6

Summary and conclusions

1. SUMMARY OF FINDINGS. 2. OVERALL CONCLUSIONS. 3. IMPLEMENTATION AND WIDER APPLICABILITY — Target Markup Adoption (TMA) — Smart Insertion (SI) — Moving blocks of text — Backspacing and deleting — Feedback when inserting new material — Adding space — The toolbar — Editing — Unstructured editing — Markup characters — DTD/Schema compilation and use of stylesheets — Appearances — Default option — Mathematics — Wider implications. 4. RECENT CHANGES AND FURTHER WORK — Historical note — Recent changes — Further work.

An experiment is a failure only when it also fails adequately to test the hypothesis in question, when the data it produces don't prove anything one way or another. (Pirsig, 1974, Part)

In examining the results, we have tried both to isolate the specific functions that could benefit from modification, and to identify those functions for which there is broader scope for more general applicability elsewhere in the interface.

In the course of this study, some changes have already taken place in the field, and others long forecast have failed to materialise. While this is in the nature of any examination of technology, it is interesting to note that only a few changes of any significance occurred in the early part of the study, but a number of important ones appear to have started in the most recent period, including a direction for the underlying file format which is as welcome as it is unexpected.

Whatever format eventually dominates, there will remain a need for an interface to it, whether it is typed, clicked, spoken, or sensed in some other way. Making

that interface appropriate to the users will therefore remain an important feature of design and implementation for structured document editors.

6.1 Summary of findings

The objective of this research was to determine if there are aspects of the software interfaces which militate against optimal document construction by writers who are not computer experts, and to suggest possible remedies.

The target users were defined as writers and editors who were not experts in computing, markup, or technical documentation, but who had a requirement to produce structured documents (limited here to XML or LaTeX). The target document types were taken as those using a formal structure for business, administrative, academic, or research purposes, and were not ephemeral, transient, or inconsequential.

From the responses to the user survey and requests analysis we identified twelve functions in the conventional editing interface to structured documents. There were many other functions for related topics which could not be included for reasons of time or scope. All the functions selected were represented in the survey and analysis as being problematic, either because their behaviour was inconsistent, unexpected, or in conflict with the structural document model; or because (as traditionally instantiated) they did not elicit sufficient information to be usable for guiding the structural formation of the document.

The four investigations (experts, software, requests, and users) revealed a number of key constraints, of which the most important were:

1. Experts felt that existing software was often inadequate to the task of providing the interface the users expected, leading to greater training and customisation costs;
2. The software, in general, did provide the features and functions required, but these were often hidden too deep, or labelled in terms that were not understood by the [potential] user population;
3. Requests for editing software showed that WYSIWYG, ‘easy-to-use’ editors which created a formal markup structure were the prime requirement;
4. The user survey provided a list of individual items in the interface that could be seen as problematic.

This information was used as a filter to split and combined the items in [4] to create a shortlist that could be tested for differences in usability compared with traditional affordances. The tests were conducted using paper prototyping, and showed that a small additional level of automation could produce acceptable improvements in usability while enabling the editing system to use or gather sufficient information to maintain the construction of a structured document. The specific test results were:

1. **Create a new journal article:** Providing pre-formed templates for different document types accessed by one of the New functions is a viable alternative to the File | New menu. While the distinction is not immediately apparent, the difference in intention is designed to discourage the creation of arbitrary-model documents (see section 6.3.13 on page 359 for a brief discussion of this mode).

   Systems would need to provide acceptable templates for all the document types a writer would reasonably need in his or her field, with others downloadable when needed.

2. **Add a new paragraph:** The New function performs the task adequately, removing the need to position the cursor, but in most cases is not a sufficient advance over the Enter key.

   The behaviour of the Enter key, however, would need to be carefully defined in a new interface, to avoid trespassing on expected behaviour.

3. **Split a paragraph:** The difficulty reported by some testers implies that this is not seen as a structural task or as any different from inserting a premature linebreak within a paragraph, and therefore does not require any additional function.

4. **Join a paragraph to the preceding one:** In this test and the next, the function was not sufficiently well-defined, nor perceived as frequently-required enough to need a dedicated function, when the Backspace or Delete key is enough.

5. **Join a paragraph to the following one:** However, there is an interesting dichotomy between going to the start of a paragraph and using Backspace, compared with going to the end of a paragraph and using Delete, which might bear investigation from the point of view of efficiency or productivity.

6. **Add a new section:** The New function appears to work well as a means of

adding a major structural item (hierarchy) without the need for manual positioning.

User habits of manual repositioning and 'creating space' for a new element (derived from wordprocessor systems) may be very deeply ingrained.

7. **Add a new list:** The New function also appears to work well as a means of adding a minor structural item (pool) without the need for manual positioning. The modified behaviour of the List toolbar buttons also appears acceptable, but may require familiarisation.

8. **Move a block of text:** The traditional cut-and-paste method appears to be well-established for short stretches of text, and for short-range moves, where scrolling is not difficult. The Mark function is one alternative for avoiding the highlighting scroll problem; the navigationally-based Move function was not easily testable in a paper prototype.

9. **Highlight a product name:** The differential detection of intent on inline markup, based on the font controls (B/I/U/S buttons), seems to work without problems after the first initial surprise at seeing a menu where no menu was expected.

   A problem — as with the provision of document types and stylesheets — is that the DTD or schema in use must be analysed to determine exactly what markup is to use this feature.

10. **Add a cross-reference:** The cross-reference button is not an established affordance (most systems appear to use the Insert menu), and the pop-up navigation window is not widely implemented, although it does now appear in *Word*. The combination appears to be a viable solution.

11. **Add a citation:** A similar argument applies to the insertion of citations and the pop-up document list, where the Cite button is well-established but the in-editor document list is not. While both this and the preceding test share a common mechanism, observation seems to imply that neither is as widely used as might be expected.

12. **Insert a fragment from another document:** The same concept applied to inserting a document fragment indicates that this is a usable method of avoiding the excise of opening the second document in another document window and doing a manual copy-and-paste.

6.2 Overall conclusions

We conclude that changes along the lines of the foregoing list are likely to contribute to making a structured-document editor more usable by the target population, while allowing the software to use the metadata gathered from the interaction to help in the construction of a structured document.

In addition, a number of unexplored techniques were identified which could reduce user frustration, among them Target Markup Adoption and Smart Insertion.

The benefits likely to follow from these changes are greater user satisfaction (because of better feedback and ease of use) and less training (because of improved usability). This would provide greater accuracy in document construction with less effort, which could contribute to improvements in effectiveness and efficiency, all of which is consistent with the goals of the ISO usability standard (ISO 9241-11:1998, 1998).

It was not possible to test directly whether these benefits would lead to lower costs within an organisation, as there are other factors which affect this, including the organisational change which may be required to implement different software. However, as several of the changes suggested have recently been tested or introduced in some interfaces, including the navigation-based access to cross-reference targets in *Word*, and the possibility of semantic drop-downs for formatting in *Xopus*, there appears to be a willingness among manufacturers as well as users to use them.

6.3 Implementation and wider applicability

As explained in section 4.2 on page 218, we could not attempt to write an entire XML or LaTeX editor from scratch within the scope of this research. Nevertheless, it is important to examine some details of how the foregoing changes could be implemented in order to identify their applicability.

All document editing software beyond the plaintext editor must by definition follow the same broad outline of operation:

1. Select and open a file;

2. Read the file into an in-memory structure, checking that it meets some criteria for valid construction, in order to gain access to the structure[1]
3. Present the content to the user in an editing window, formatted by reference to a stylesheet;
4. Operate a model of interaction to allow the user to enter and change the content, structure, and appearance;
5. Save and close the file, either automatically or on command.

We have not been concerned here with the detail of parsing and the exposure of the **grove** (SGML data-model) or the Post-Schema-Validation Infoset (PSVI) (XML data-model) to the program, as these are standard operations provided by well-established libraries of software; nor have we addressed the saving and closing of files, which is a straightforward mechanical operation.

We have, however, examined file selection and opening, as defined in section 4.3.2.1 on page 244 and section 4.3.2.2 on page 245, and we have discussed extensively the presentation of content for editing with reference to the use of stylesheets *passim*. Our principal concern has been the model of interaction for editing, for which we have identified some design patterns we consider the most conducive to the implementation of the tasks.

In the sections that follow here, we present some of the observations and deductions made during the course of the research, relating specifically to how the design patterns could be implemented in an interface (in section 6.3.1 to section 6.3.14). In section 6.3.15 on page 360 we consider the implications of these patterns for other applications.

6.3.1 Target Markup Adoption (TMA)

This is a set of principles to help determine how a fragment of text should be handled when copied and pasted from a domain outside that of the document (for example a web page or another document using different markup). It can also be applied to text pasted from another context within the same document.

The problem is evident in existing systems where the precise formatting of the source text is painstakingly maintained when the fragment is pasted into the

[1] As noted in item 4 in the list on page 282, an invalid or well-formed file may still be opened, for repair, albeit in an entirely different type of interface.

target environment, regardless of the surrounding material in the target. This is almost always wrong.

Such copying and pasting should probably never preserve the source typeface, size, and layout *in mixed content*, as this will rarely be the same as the one in use in the target document (pasting whole subdocuments as self-contained blocks such as figures may possibly justify retaining the styling). Generic visual indications such as bold and italic should however be preserved, and should adopt their counterparts in the target environment. Other typographic features such as margin widths and line-spacing should be entirely dropped from the source, and those of the target adopted.

When pasting fragmentary mixed content, the text should therefore adopt the style of the target container and not the source style, subject to the mapping of sub-element types or style-detection. When pasting whole elements in element content, however, the internal consistency of styling should be retained, subject to any inheritance or disinheritance at the target location: if subsubsections in the target document are formatted in smaller type than normal, an alien subsubsection-equivalent pasted from elsewhere should adopt the smaller font.

Where copy and paste is done from documents using the same DTD or Schema, markup alignment is at least manageable, if not entirely straightforward; but where no equivalent target markup is known or immediately detectable, a dialog would be needed to request the mapping from the user. The 'pre-flight' semantic assignments used by *Arbortext Architect* (see section 6.3.11.1 on page 356) would go some way towards avoiding this, and there is some scope for educated guesswork — while it might dismay some purists (see footnote 3 on p. 356), one DTD's `para` element is very likely to be intended for the same purpose as another Schema's `p` element.

This principle is already in partial use in some embedded XML editors designed for web applications such as *TinyMCE*, and can be used to sanitise Word markup before pasting. It does however need some foreknowledge of the default element styling applicable to a given location: in a structured document editor this should already be available from the DTD or Schema and the stylesheet.

There are of course cases where this technique must *not* be used. If the document is about document design, for example, there will almost certainly be places where examples in a specific set of styles need to be reproduced intact, without interference. A similar rule might apply to the reproduction of lemma and

witnesses in literary analytic editions.

6.3.2 Smart Insertion (SI)

This principle is to structural markup as Target Markup Adoption (TMA) is to formatting. In common with TMA, it deals both with material pasted from outside the document scope and with material from inside. To overcome the problems associated with element insertion in element content, when the user's cursor is positioned elsewhere, particularly in mixed content, it is essential to adopt a benevolent stance.

We first consider the pasting of whole elements (in the XML sense) from the same DTD or Schema.

When the target location is in mixed content, and the insertion is an element type that is *not* in the content model at that point, the content model in the parent element is checked to see the if the requested element is valid there, and if so, make the insertion there, *after* the target location (unless the content model determines otherwise).

If the element was not acceptable in the parent content model, ascend the hierarchy, testing at each level until a location is reached where it is valid, and insert it there instead, again *after* the target location where possible.

If the ascent reaches the root element without locating a valid point for insertion, identify the possible locations of the requested element type in the content models provided by the DTD or Schema, and compare their ladder locations by descent with those already in existence in the current document instance. Pick the shortest match for a potential location occurring *after* the original target location where insertion was requested, and make the insertion there, recursively creating any intervening container elements necessary to preserve validity.

Pasting arbitrary fragments is a special case of this principle, and is dealt with in section 6.3.3 below.

6.3.3 Moving blocks of text

Wordprocessors and other editors operating the 'anything anywhere' document model allow arbitrary cut and paste regardless of whether the fragment crosses

what would normally be regarded as structural boundaries. The formatting of the source is preserved regardless of the target location.

Structured editors, on the other hand, restrict the marking (highlighting) of material that crosses structural boundaries: for example, one cannot click and hold down the mouse key in the middle of one section and drag the highlighting across the end of the section, over the next section heading, and into the next section. This is done to avoid the resulting fragmentary structure of the pasted text breaking the rules of the document type. In many systems, an attempt to do this will simply result in the highlighted fragment of the first section being de-highlighted once the cursor crossed the structural boundary into the following section. At the worst, it will simply beep or flash an error dialog.

However, user expectation is that you should be able to highlight from anywhere to anywhere, cut or copy it, and paste or drag it elsewhere without complaint. Further, the current expectation is that there will be manual cleaning-up of the formatting as a result of this, because the editor has only sparse, if any, information about the circumstances of the source of the text and the circumstances of the target location.

However, in a structured document environment, a very considerable amount is known about the circumstances of both source and target, so it should be possible to have the computer deal with the consequences automatically. While this amount of latitude may occasionally be undesirable in respect of the *meaning* of the text, we see no reason to forbid it if we apply the principles of TMA and SI.

Consider the position suggested above, where an author highlights text starting in the middle of the last paragraph of a section, and continuing over the next section heading into the middle of the first paragraph of the next section (see Figure 6.1 on the following page).

If the author or editor now performs a Cut, the abutting paragraph fragments will be joined seamlessly at the words '*passim*.' and 'The', creating a single paragraph and eliding the heading, so that the remainder of what was section 6.3.1 now forms the end of the pre-sectioning introduction to section 6.3 (the boundary at the end of the former section 6.3.1 would be removed and any nested sections promoted).

This operation is impossible in most structured systems because it would create a temporary state of affairs where the two abutting paragraphs have been merged, but the loss of the structural boundary for the start of section 6.3.1 means the

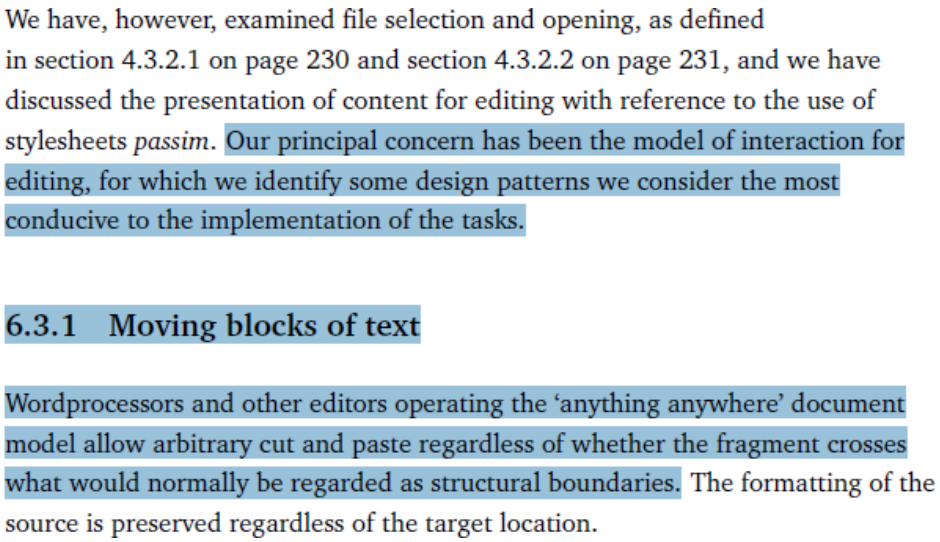


Figure 6.1: Arbitrary highlighted text

boundary at the end of that section is now orphaned. Losing this boundary might also create a state where any subsections of 6.3.1 were now in conflict with the document model (which is why they would need promoting to a higher level).

Let us now consider what happens when the text thus cut is pasted into a new location.

The text begins with an implicit paragraph-start, because it was cut from mid-paragraph (if the paragraph had been cut whole, it would have an explicit paragraph-start identity). If the target location is in mid-paragraph, well and good: the start and end can be merged at that point. If the target location is between paragraphs (or other elements in element content: the pool), the the start of the fragment must acquire a paragraph-start status, and the end must acquire a paragraph-end status.

The intervening section break must then be considered with respect to the level of the target location in the document hierarchy. If it is a section of the same level, the pasted section break is simply honoured as-is, and becomes the start of a new section at that level, with the relevant numbering (and subsequent sections are renumbered accordingly).

If the fragment is pasted into a section of lower level, there is a choice: the section break could become another at the same level, or at one level lower still. This would need to be governed by a default rule: we suggest that it may be considered less likely that an author would intend demoting the fragment yet further, so choosing to create a new section at the level of insertion should be the

default (with the renumbering of following existing sections at that level).

If the fragment is pasted into a section of higher level, however, we consider the reverse should apply: the pasted section break should probably become a lower-order section, and any following existing sections at the lower level renumbered.

There is clearly scope for much testing and evaluation here, but the principle appears to be sound, and in accordance with the idea of Smart Insertion, that the inserted material should adapt to the constraints of its surroundings. The exemptions mentioned at the end of section 6.3.1 on page 344 would not apply here, as the target markup *structure* must accord with its DTD or Schema, as opposed to its *styling*, from which the exemptions could be, well, exempt.

6.3.4 Backspacing and deleting

We have seen from the tests that there was a minor confusion observed in users over the expected effects of the Backspace key; and a possible preference to use the Delete key when joining a paragraph with the next one, and to use the Backspace key when joining a paragraph with the preceding one. This did not, however, relate to other markup, or to the deletion of boundaries to a higher or lower level.

In a WYSIWYG environment, where no markup is displayed, and its presence can only be inferred from typographic styling, all text is regarded as peer. In common with the attitude to cutting text, deletion across a boundary appears from observation to involve no sense of the boundary existing. Navigational problems of this sort have been reported for a long time (eg Payne (1991); Dau and Sifer (2007)), and have led to at least one (now obsolete) software product explicitly designed to over come them (*Xeena* from IBM Alphaworks), but the problem is viewed as intractable if the markup is hidden (as required *a priori* in this research).

This experimental procedure uses Backspace: directions can be reversed for use of the Delete key. Refer to Figure 6.2 on the following page for a decision-tree. An attempt was made to model the procedure using the Task-Action grammar formalised in Payne and Green (1986), but the need to navigate around the markup added a level of complexity that was inappropriate in such a small scale example.

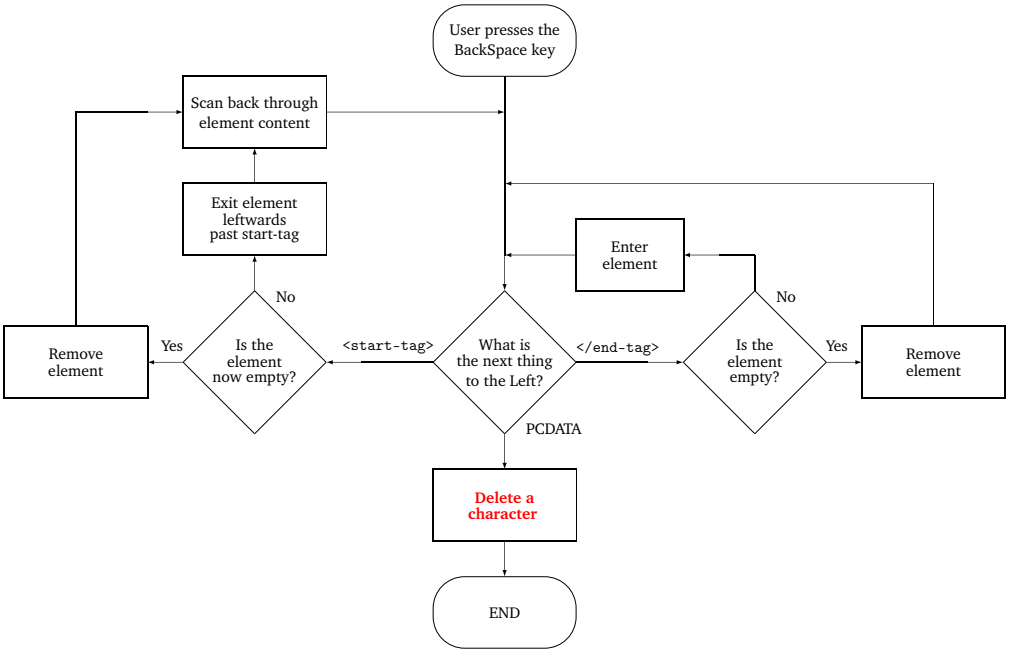


Figure 6.2: Deciding how to handle markup when deleting

Procedure 6.1: Handling markup boundaries while deleting text

1. If the character to the left of the cursor is normal character data content, delete it.

2. If there are no more characters of normal data content remaining to the left of the cursor in the current element:

   (a) if the element is now empty (no remaining character data content to the right of the cursor either), then remove the empty element if this is permitted by the DTD/Schema. If not permitted, retain the empty element but move the cursor leftwards to stand immediately before it.

   (b) if there *was* character data content remaining in the element (to the right of the cursor), move the cursor leftwards to stand immediately before the element.

   In the case where the cursor now stands in element content, move the cursor into the next preceding element with a content model including character data.

3. If the cursor is immediately to the right of an element end boundary, move the cursor into the next preceding element with a content model including character data, and continue deleting

leftwards within that element.

When such an element's content has been entirely deleted, the element markup itself should be silently removed and deletion should continue leftwards if there is a further sibling text node or element present.

An EMPTY element type should be deleted *if it has a visual representation* (such as generated content) *as if it were a single character*. If it has no visual representation, then it cannot be seen by the user, so it should be deleted with a signal being given to the user.

It will be seen that the concept has many similarities to the moving of block text, and could perhaps be termed 'Smart Deletion'.

6.3.5 Feedback when inserting new material

Among the comments made in the Expert survey was a general condemnation of software that goes 'beep' when a key is pressed or a menu item selected (or which greys out the option) in a location where its function is not permitted by the document structure *when the function could reasonably be expected by the user to be available*.

An elementary example was evident in the SGML editor *Author/Editor* and its XML successor, *XMetaL*, where pressing the Enter key in mixed content produced a beep and a space (the beep was later silenced). This happened because in mixed content, a premature line-break required markup to act as a placeholder (for example BR in HTML): a simple newline was regarded as merely character data, and not significant as markup. The untrained user would be frustrated because the expectation is that pressing the Enter key will create a new line.

A more frequent example is an attempt by the user to insert an element type in a place where it is not permitted by the DTD or Schema. The reader can model this in almost any XML editor:

1. Select a document structured with chapters, sections, and subsections.
2. Place the cursor in mid-paragraph in a section or subsection.

3. Try to start a new chapter.

The editor will know from the DTD or Schema that a chapter element is not permitted in mid-paragraph, so the function will simply not be available: either a warning message will appear, or the chapter element type will not be available in the menu. The design pattern employed here appears to be that it is considered the user's responsibility to know where the cursor is at any given time, and to move cursor to the correct location (after the end of a chapter) before a new chapter can be inserted.

This raises a secondary problem in interfaces where the markup is hidden: in the case of *appending* a new chapter (rather than inserting one between existing chapters), the location 'after the end of a chapter' may be invisible, and the system may not allow the cursor to be placed anywhere except after the full point at the end of the last sentence. In effect, the location in element content for a new chapter is rendered inaccessible by too strict an adherence to the principle of hiding the markup. In an extreme case (described in section 4.3.5.3 on page 267) this resulted in eleven entire chapters being pasted into a footnote, and deleted when the footnote was later removed!

In these circumstances, the user is expected to understand why the function is inapplicable, which in some cases involves more foreknowledge of the DTD or Schema and its construction than can reasonably be expected, sometimes even from an expert user. The model described in section 6.3.2 on page 344 offers an alternative for handling the Insert function: it should not be necessary to expect the user to know as much as the expert!

6.3.6 Adding space

We saw in section 4.3.1.3 on page 238 that the use of multiple spaces was a common 'solution' to the user requirement for wider spacing or a deliberate blank gap. There are several markup solutions to this: a Processing Instruction containing the measure; a symbolic element type such as TEI's `<gap/>`; a repeated character entity such as HTML's ` ` ; or a 'literal' element type for which the expected output would preserve the white-space, such as HTML's `pre` or DocBook's `literal`.

If the keypresses for multiple space, newline, or tab are detected, one of the above solutions should be included in every DTD or Schema where users may wish to

implement it. It then becomes relatively simple to pop up the relevant option depending on which key has been pressed multiple times.

However, the use of a container element type whose contents are multiple space or newline characters is not robust, as wrapping and folding in an insensitive editor may destroy the white-space, and there is no way to ensure that subsequent processes (for example, XML) are executed with the `xml:space` control set to preserve the spacing.

6.3.7 The toolbar

While all the controls of an editor are contained in the menus — somewhere — a toolbar presents a selection of the most commonly-required controls, as imagined or deduced by the authors of the program, or as configured by the user. In some cases, controls can be removed from, or added to, the toolbar in order to provide exactly the set required for a specific task or by a certain person. Figure 6.3 illustrates this activity using a drag-and-drop dialog.

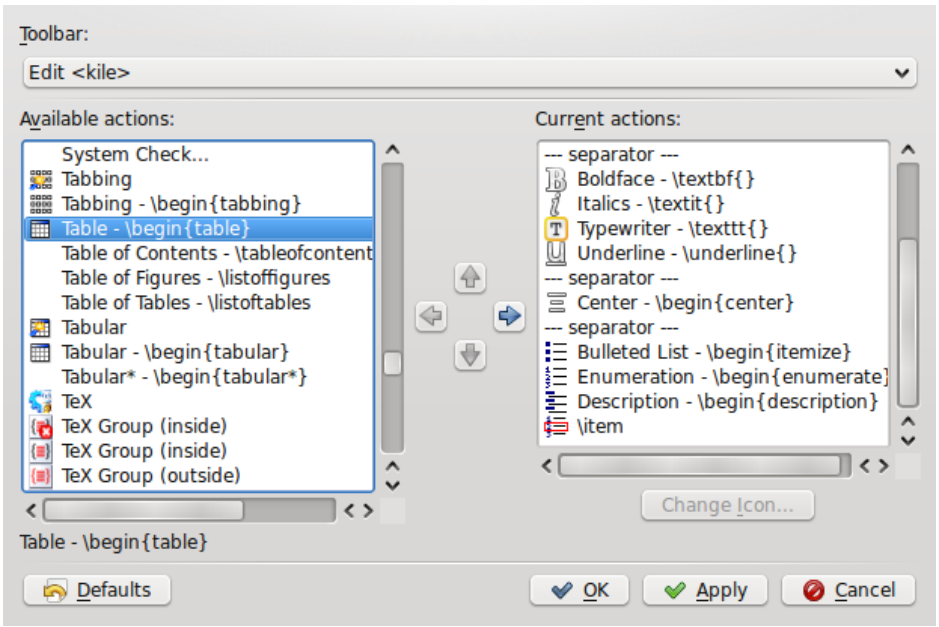


Figure 6.3: Adding an item to the toolbar in the LaTeX editor *Kile*

The typical default toolbar in an editing application contains a large number of standard controls common to all systems — specifically, those omitted from the scope of this research because they do not relate directly to markup — and a smaller number of controls which map to some of the items in this list.

The affordance provided by a toolbar icon is greater than that provided by the same control in a menu because it is already visible, whereas the menu option needs to be discovered. As many toolbars can be configured with the name labelling of the items turned off for compactness and graphical appeal, the icons used for the controls need to have a high degree of obviousness for them to be acceptable as affordances. When the affordance of the toolbar icon is insufficient, the user will need to spend time finding the control in the menus, or seeking some other way to achieve the desired result.[2]

Placement and presence of items in the toolbar should be determinable by the user, as is already the case in some systems (assuming that the user is amenable to using toolbar items: some users will prefer to use keyboard shortcuts or other methods). Automated adjustment according to their frequency of use by the user (as originally envisaged for the Microsoft Office 2010 'ribbon') may lead to confusion, and makes training almost impossible if no one user's toolbar is like any other's. However, the additional features we tested in this research were, we believe, sufficiently important to warrant their inclusion, so the test harness therefore had them in the toolbar by default.

6.3.8 Editing

A number of editorial control functions were seen by respondents as both desirable and problematic (section 3.4.3.13 on page 201). Chief among these was the cut-and-paste action on large quantities of material such as a whole section (the Block move function). This was seen as poorly handled by most software particularly when it was combined with the need for promotion or demotion (moving a section to a subsection position, or *vice versa*; or changing list types). We believe we have dealt with this in outline in section 6.3.3 on page 344.

There are two aspects to this: firstly, the system needs to be aware of the relative structural implications of the element types used for containing the hierarchy; for example, it needs to 'know' that a `sect2` element can only occur in the element content of a `sect1` element, and then only immediately before an existing `sect2`, between adjacent `sect2`s, or at the end of the containing `sect1`, for reasons described in the fourth item on page 12 and the associated Figure 1.6 on page 13.

[2] A related feature of *Emacs* that contributes to user education is that when a command is typed in the mini-buffer, for which a keyboard shortcut exists, the shortcut mapping is displayed briefly as a reminder to the user.

This information is provided explicitly in the DTD or Schema for XML documents but is not addressed at all in LaTeX. However, not all editing systems are yet capable of using this information for purposes of promotion or demotion of elements in hierarchical element content.

Secondly, the substructure of the block being moved must also be checked and accommodated, as we have shown in section 6.3.2 on page 344; that is, the nested markup in both element and mixed content must be accepted at the point of insertion, or modified according to the technique described in section 6.3.1 on page 342.

Again, it must be emphasised that we are considering that rejection is not an option: a system that beeps or flashes when a cut-and-paste of this nature is attempted is likely to be shunned by the user.

Editing and markup across element boundaries.

6.3.9 Unstructured editing

A number of editors allow the user to create a simple well-formed XML document with no DTD, by detecting the typing of a start-tag and automatically providing the matching end-tag (*XED*, for example (Thompson, 2000)). This technique can be very convenient on some occasions: the expert who wants to test a particular design without having to write a DTD, or who wants to be able to open a file quickly and make a change (in the knowledge that she knows what she is doing); and for any user who needs a trivial instance of a document for test or demonstration purposes.

Wordprocessors are also inherently unstructured editors: they produce a flat file consisting of paragraphs (this is essentially true even if the paragraphs are encoded in XML, as with *Word* and *OpenOffice*). Attributes on these paragraphs can identify formatting styles such as ‘title’ or ‘list item’, but that does not change the structure, only the appearance (the sole exception is noted in footnote 8 on p. 35). DeRose (1997, p. 109) identifies at least five problems created by this flat structure:

- block moves and outliners cannot promote or demote;
- tables of contents may select the wrong levels;
- ‘keep-together’ functions cannot be extended beyond one following element;
- nested numbered lists may fail to follow sequence;

- more complex list structure may not be supportable.

It would be expected that a large quantity of programming effort is contained within wordprocessors to accommodate this fundamental deficiency, which (it must be assumed) derives from the [perceived] market demand for a system which allows the author to put anything anywhere, regardless of whether or not it makes any logical sense.

Deducing a DTD from a non-rigorously-composed document instance is many times more difficult, but has been possible since the early 1990s (the *Fred* application from the Online Computer Library Center (OCLC); see Keith (1995)), and a similar facility is now available within many XML Interactive Development Environments (IDEs). Such a generated Schema or DTD applies only to the specific instance from which it was deduced, however, and requires considerably more work to generalise it for an equivalent class of documents.

The addition of such a feature to a structured-document editor seems at first sight somewhat perverse, but the evidence of comments to the messages reviewed in the Requests Analysis (section 3.3 on page 161) is that many novice users do indeed expect to be able to sit down and start typing immediately, as with a wordprocessor, and have the system guess (correctly!) what they are trying to do. As is mentioned in section 6.3.13 on page 359, it is claimed to be possible, given a sufficiently large set of knowledge, accurately to deduce intentions from typeface, font, screen position, and other aspects of the typed text; but it is unfortunately outside the scope of this research to investigate this.

6.3.10 Markup characters

In the display model we propose, there are no markup characters (pointy brackets, ampersands, backslashes, curly braces, etc) visible at any time except in a specifically-requested 'reveal' or 'tags-on' mode. This implies a robust and comprehensive stylesheet for each supported document type so that editing via visual means can be fully supported: see section 6.3.11.1 on page 356.

The effect of element markup in element content (XML) and sectioning or outer environments (LaTeX) is generally non-problematic, as its representation normally follows well-established patterns (chapter starts, section heads, paragraph breaks, list indents, etc), so there should never be a need to see the markup in the display.

For many common element types in mixed content, a similar argument can be

made for the formatting of simple typographic variants (bold, italics, etc) and for the highlighting of [cross-]references.

However, character entity references (XML) and specialist graphics (LaTeX) normally require to be resolved by a strict adherence to Unicode and the provision of suitable fonts. In these cases, a symbolic representation of the object may be needed when a suitable font is not available. The problem as occurred elsewhere (web browsers) and is already largely solved by the *Gecko* rendering engine used in the *Firefox* browser in similar circumstances, so the same techniques could easily be applied to editors.

Special exceptions will always occur, and there are a number of possible models for the representation of two major classes where evidence of the markup may not be sufficiently visible: *a*) non-Unicode symbols; and *b*) deeply-nested (possibly conflicting) inline markup. The first is an edge case (Cooper, 2004, p. 100), a classic borderline condition (Knuth, 1986, p. 255) too rare to consider here. The second class is a matter of some considerable research, notably in the Humanities, where the TEI markup allows exhaustive tagging of great minuteness (Fenton & Duggan, 2006).

Ultimately, some form of symbolic, graphical, or typographical representation of markup needs to be shown when the user specifies an otherwise unformattable object, ranging from a broken image file to a piece of markup for which no styling rule exists. A system would need to prompt for a specification of how they would like it displayed, or pop up a styling window, and commit the decision to the current stylesheet for re-use.

6.3.11 DTD/Schema compilation and use of stylesheets

There is currently no central repository of all known document types, although there are several partial collections organised on a sectoral basis in different industries, markets, languages, cultures, countries, and research areas. The addition of a simple resource management function to allow the user to add, update, and remove the URIs of these collections would be of benefit to users who frequently need to compile new document types, but this is an implementation detail outside the scope of this research.

When compilation (or whatever digestion process the editor requires) has taken place, the resulting internal image should be stored with a title or name label in

the manner of other precompiled document types, and made available by the same mechanism for future use. It is remarkable that many structured editors happily compile ad-hoc document types but fail to make them usable simply for new documents of the same type.

6.3.11.1 Stylesheets

A critical component of such precompiled document types is that they are shipped by the vendor with stylesheets suitable for displaying the typographical format necessary for WYSIWYG editing. One recurrent theme noticed in the Requests Analysis (section 3.3 on page 161) was the puzzlement of users that editors did not already 'know' how a particular document type should be displayed, as if there was some magic built into in a DTD or Schema that would allow the 'correct' formatting to occur without human intervention.

This may be partly due to an assumption that because there appear to be inherent implied semantics to element type names in many document types[3] that are 'obvious' to a human, it should therefore also be obvious to a computer program, or at least be guessable: an element type called `para` or `itemizedlist` can not unreasonably be deemed to be 'obvious' — if you come from an English-speaking westernised culture. However, structured editors do not do this, possibly for sound reasons of semantic misinterpretation, although this author is unaware of any published research into the topic.

A partial solution to the problem has been implemented for many years by the *Arbortext Architect* document-type compiler. On completion of the compiling operation, the user is presented with a sequence of dialogs asking for the identification (element type name) for the following components:

- document title;
- enclosing title block;
- title page;
- paragraph;
- graphics (with attributes for positioning, cropping, and scaling);
- touchups (minor post-typesetting adjustments);
- sectioning (with levels and title element);

[3]The denial of semantics in markup names is an interesting phenomenon. The reader is referred to the comments of Len Bullard and Ken Holman at `http://lists.xml.org/archives/xml-dev/201310/msg00057.html`.

- list items (numbered or bulleted);
- definition-list items;
- pagebreaks.

This method of collecting what are presumed to be the document type designers' intentions for the use of key element types allows the software to build a rudimentary stylesheet, so that the editor can provide a workable skeleton display format right from the start of the first document instance. It does, however, rely on the user having a fairly comprehensive foreknowledge of the document type; this can be perhaps be assumed in the case of the target market for *Arbortext Architect*, but it cannot be assumed at all for our target users here.

The list is in any event somewhat eclectic, having been derived from the vendor's experience in the technical and military documentation field: it would be useful to extend it to cover author, date, abstract or summary, block quotation, table, figure, appendix, and cross-reference element types, and perhaps more. However, this would place it well beyond the capabilities of non-XML-experts to identify the element type names, although there is nothing to stop the software making some key assumptions based on dictionary or thesaurus interpretations.

6.3.12 Appearances

It has become conventional for writers to create headings, titles, and other textual features by applying a change of typeface through highlighting the text and using the font and size dropdown menus. In an environment where styles are in use, the author or editor highlights the text but then selects the relevant style name from a menu, and the correct typeface, size, and spacing is applied automatically from the stylesheet.

A solution to the problem of achieving persistent styling in the WYSIWYG display is also implemented in many editors. With the cursor located in a given element type, the styling dialog can be invoked from a menu item or toolbar icon, and the typeface, font, size, weight, spacing, and other parameters selected. This can be saved for global application to all documents of this type for any occurrence of the selected element type, optionally with an XPath or similar location ladder to control variant appearances in different contexts. The *Author/Editor* editing software and the (derivative) *Panorama* SGML browser plugin for Netscape *Navigator* were two excellent early examples of this, as is the stylesheet mechanism in *WordPerfect SGML* and *Arbortext*; a similar although less

sophisticated tool is in the *epcEdit* editor.

As we have seen in section 1.1.2 on page 8, the application of styled markup can serve two purposes: identification styling as well as structural styling. The styling can be for authorial purposes (structure, visibility, or even a reminder to return to this point later for further work); or for editorial purposes (structure again, indexing, referencing, and of course formatting). When applied robustly and consistently, however, the most effective use is for identity so that a transformation to a typesetting environment can automate a large part of the process.

Selecting a typeface or size or style should therefore apply that formatting to the highlighted text as the user would expect, but would also record the action in the stylesheet. A dialog can be used to ask if that element type's style should be changed to reflect the formatting throughout the document, or only in this one circumstance. A similar approach was taken in this research in Figure 4.4 on page 234. In this way, the author can trivially build up a working stylesheet merely by formatting things as she has previously done, and allowing the software to propagate their applicability.

6.3.13 Default option

In the absence of any workable method of deducing or identifying the [approximate] placeholder formatting required for the elements of a document type, the only acceptable assumption that can be made is that element types with mixed content (PCDATA and subelements) which themselves occur only in element content are very likely to be paragraphic in form, and can therefore initially be styled as blocks. This and other assumptions can be described in a form of state table as shown in Table 6.1 on the facing page.

In this author's view, there is scope for some valuable work to be done on the detection of default intent in this manner, although there will always be room for error.[4] There are many problems, not least in the [mis-]application of the method to Schemas intended for data storage rather than text, and in the

[4]Both the TEI and DocBook enable elements with element content models to occur in mixed content. DocBook does this to enable computing structures to be embedded at paragraph level; the TEI does so to enable even entire document texts to be embedded in a paragraph, as occurs in some literary and historical texts. HTML, by contrast, allows character data almost everywhere, even in the so-called 'Strict' versions of XHTML.

Table 6.1: Initial formatting assumptions deducible from a DTD or Schema

	Declared content		
Container	**Element content**	**Mixed content**	**PCDATA only**
in Element Content	Block	Paragraph	Label or heading?
in Mixed Content	Substructure occurring inside a paragraph: not easily guessed	Inline	Inline

Referenceable element types such as cross-references may be discernable by the presence of ID or IDREF attributes.

non-applicability of the method to document types in which there is no mixed content at all, such as OOXML (all text being fragmented in element types with PCDATA declared content); but that should not prevent software designers from at least providing the user with a 'first-draft' stylesheet which can later be amended as described in section 6.3.12 on page 357.

In DTDless mode, the detection of intent would need to be based on a combination of the analysis of a writer's arbitrary formatting and the names or typology applied to element types created on-the-fly. Neither approach has been tested in a structured-document environment so far as can be ascertained, although Novell did once produce a printer-driver substitute that allowed wordprocessor documents to be 'printed' to software which would attempt such disambiguation based solely on typographic formatting; and software now exists to perform a similar task on unstructured PDF documents in an attempt to recover their structure.

6.3.14 Mathematics

We have referred earlier to the detail of mathematics editing being out of scope for this research, as it is a field of great depth and complexity in its own right. Mathematical expressions should — under our terms of conduct — be constructed graphically as far as possible; but some form of access to an internal symbolic coding (possibly LaTeX) will probably always be required for expert use.

As explained in 'Mathematical markup' in the list in section 3.3.5 on page 168, the most extensive graphical mathematical formatting for structured editors is provided by the Arbortext editor (for XML) and by LyX (for LaTeX). Both are

remarkably easy to use even by occasional or rare mathematics users (such as the present author!).[5]

It would appear reasonable to expect MathML (the standard, but vast and complex, markup vocabulary for mathematics) to be enabled in any DTD or Schema likely to be used by mathematicians (it can already be combined with DocBook and with TEI). It is therefore clearly possible to provide a more limited mathematical markup vocabulary in other XML environments where a simple `math` element type is enabled. Invoking mathematics could therefore default to a simple visual mode for expressions whose formatting is trivial such as $E = mc^2$ and provide an optional power-mode for the expert user, which would also offer a configurable choice between the semantic and visual modes of MathML. In other circumstances, software needs to allow the importation of an external image, or the use of TEX/LATEX syntax in an attribute value. In a LATEX editor, by its nature, no special facilities are needed, as the rendering can be provided in near real time by packages such as preview-latex (Larsson & Kastrup, 2007) and Instant Preview (Fine, 2001).

6.3.15 Wider implications

Apart from some expressions of surprise and understandable confusions over terminology, most of the testers were curious to know when 'it' would be available; 'it' being the tested interface that they presumed was imminently about to be released. Such confidence is encouraging to the researcher, but it implies that within this small sample drawn from a single institution, there would be interest in working more efficiently, based on the modifications proposed.

Bringing such changes to market is a much larger question, and one that can only be resolved by the editor-writing companies and projects. We have discussed

[5] If there were a prize for achievement, however, it would certainly have been awarded to the forgotten authors of the EuroMath project's editing software from the early 1990s. This program, available at the time only for Sun workstations, allowed mathematicians to write their papers in SGML without knowing it, using a wordprocessor-like graphical interface; and to enter mathematical expressions of arbitrary complexity in TEX notation and see them displayed in typeset form — and then have them transformed with a single click to SGML markup. Even more remarkably, it was possible to convert from SGML format to TEX format and back again repeatedly without loss. So far as this author has been able to ascertain, this has never been done since, although a more recent renascence of a currently dormant project with the same name on the SourceForge web site appears to have this as one of the goals.

above some of the technical hurdles to implementation, and some of the implementations that have already been done.

The general implications of these results, however, are that the majority of these changes do resolve some of the problems of capturing sufficient metadata about the authors' intentions to enable it to be used in creating a structured document. As we mentioned in section 2.1.2.2 on page 63 and elsewhere, many of the ideas behind the 'Web 2.0' interface paradigm are based on the core ideas behind UCD. The capture of intent, the minimisation of excise, the removal of unnecessary steps, and the tidying-away of visual clutter, are key stages on the road to improving an editor interface just as much as they would be in improving a web interface.

The changes we tested were largely conceptual: there was very little physical change to the interface layout beyond the addition or renaming of some buttons or menus. Three concepts in particular stand out:

1. The use of a drop-down menu when an apparently monodic button is pressed (B, I, U, etc). We have argued that the affordance of these buttons is such that their appearance cannot be changed (unless they are elided entirely, as in LyX), so it was a valid step to test the idea that a drop-down could capture the author's intent with minimal excise;

2. The use of modal pop-up dialogs to gather user intent about styling, and record it in the stylesheet for application elsewhere. We remain uncomfortable about modal dialogs, as they are intrusive, especially when unexpected. However, with careful programming — dare one say, 'Web 2.0'? — their jarring appearance can be softened; if they then demonstrate their utility, the annoyance factor should reduce;

3. The automatic repositioning of the cursor to the next available location, when a new element is requested that cannot be placed in the current location. From observation, adding new elements to the hierarchy usually requires exactly this; adding them to the pool *may* require manual intervention; and adding them to the flow almost never requires it, because it is not an ordered sequence by nature.

As to the first concept, there seems no reason why a drop-down from a button (or any other widget) should not have a persistent place in other type of interface, particularly if it reduces the need for full-blown in-your-face modal dialogs. In such dialogs, the presence of an 'Always do this' or 'Remember this choice' option

helps to reduce the impact, because the user now knows that the dialog will never return — at least in that combination of circumstances. This is already commonplace in dialogs in mobile apps.

As to the second, the difference in application in a styled interface is critical. While the prevailing method of operation in wordprocessing is to do everything manually, the idea of a stylesheet *governing* document construction is alien to most users, even though it is the current method of working underneath the interface. When a user modifies a *Word* named style, the result gets applied throughout the document. The same was true for the *Panorama* styling interface mentioned in section 6.3.12 on page 357. The difference is that so few *Word* users are aware of the existence of named styles, because the defaults are regarded as 'good enough', whereas in *Panorama* there were no defaults: a document with no stylesheet was entirely unstyled until the stylesheet interface was invoked — and we have already seen how this could be changed, in section 6.3.11.1 on page 356.

There is a facet of the document-construction problem that we did not address, namely the detection of sectional divisions based solely on visual styling, rather than on the explicit selection of a new division (via the New function). Strictly speaking, this is not an aspect of interface design, but a task in heuristic programming, where the size, style, font, and position of a paragraph (often a short line) are analysed to see if they form a pattern from which information about sectioning can be deduced.

Several systems have been used to do this: among the earliest was a printer-driver substitute from Novell in the days of MS-DOS, which attempted to reconstruct a *WordPerfect* document from the font information in the print stream. Modern systems for converting PDF [back] to *Word* or LaTeX are descendants of this, possibly backed by foreknowledge of *Word*'s default styles.

While the process has been compared with trying to turn hamburgers back into whole cows (Murray-Rust, 2008) or scrambled eggs back into chickens, I am not aware of any attempts to make it work interactively as discussed in section 6.3.13 on page 359. There would appear to be scope for using the author's interaction with the font size, style, and typeface dropdown menus to [re-]construct the prevailing stylesheet in real time, offering matches with existing styles as suggestions to adopt that style, rather than have the text remain styled on an *ad hoc* basis.

As to the third concept (automatic repositioning), some additional imagination is

required when intervening at the level of the third item above. Rapid writing or editing by a fluent expert probably needs no such intrusion (or at least could exploit it to even greater effect), but it should have place as a means of avoiding the bafflement of finding a button or menu item greyed out when the novice or occasional user least expects it. And in circumstances where correct preservation of a strict OHCO is required, such as legal documents, some governance of insertion of new material could work in conjunction with TMA and SI to ensure that the structure of the document is not violated. One can only imagine Louis Wu, editing his contracts with the Hindmost, projected on a wall of rockface in the *Ringworld* novels, benefitting from the hidden governance of a structured editing system behind his interface.

6.4 Recent changes and further work

Since the bulk of this research was carried out, there have been some updates to a few products which are starting to implement changes in the interface along the lines we have investigated. In order to compare this with past advances in the dialogue between wordprocessors and markup editors, some of the earlier activity is summarised first.

6.4.1 Historical note

There have been several systems over the years which have added SGML or XML conformance-checking and structural guidance to Microsoft *Word*. It is important to understand that recent changes are really only building on the depth of knowledge which has accumulated over nearly three decades of graphical wordprocessors and angle-bracket markup systems living side-by-side.

The first was Electronic Book Technologies' (EBT's) *Dynatag*, which let the user graphically match paragraph and character styles in *Word* files to an SGML format. Once a sufficient number of files of the same pattern had been 'trained', the program could batch-convert a large number of similar files with great reliability (Flynn, 1998, pp. 325–327). It could also convert to the Rainbow DTD, which was a popular 'half-way house' in the 1980s for such conversions; a free application (*rbmaker*) which used Rainbow was provided by EBT. The *Dynatag* product was sold with the company to Inso Corporation in 1996 but was still available in 2009 from Red Bridge Technologies, who occupy the old EBT premises.

Microsoft's own *SGML Author for Word*, despite its name, was not an authoring system, but a graphically-configurable SGML-to-*Word* converter — and back again. With care, it was capable of lossless circular conversion, allowing non-experts to author and edit in *Word*, and have the results processed by technical staff for final editing and single-source publishing (Flynn, 1998, pp. 327–331). The product was never actively marketed: indeed during evaluation in the 1990s the present author had extreme difficulty in convincing Microsoft Support in Dublin that it was in fact a Microsoft product; in its day it was widely spoken of as a 'checkbox item' designed to enable the company to bid for lucrative contracts which mandated SGML. Ironically, it was discontinued when XML appeared, just at the point when it would have enjoyed substantial sales.

Numerous add-ons for *Word* have been produced, including one from Microstar,

developers of the once-ubiquitous graphical DTD design tool, *Near&Far* (Flynn, 1998, pp. 71–74), but none of them achieved the task of shielding the author completely, either from the technicalities of XML or from the vagaries of the *Word* interface. *Word* has had its own built-in XML editor (since 2003) but it is cumbersome, restricted to W3C Schemas (unattractive to publishers, who use DTDs), and it offers no advantages in the interface over other traditional XML editors (possibly apart from being manufactured by Microsoft).

6.4.2 Recent changes

A recent XML authoring add-on to *Word* from Quark, however, explicitly modifies the *Word 2010* interface (the 'ribbon') to remove those options which would militate against creating conformant XML documents (Quark, 2012). The revised dropdown menus nevertheless still implement the element type names of the target DTD or Schema, in a similar way to the Insert function of a traditional XML editor.

The *Xopus* editor, recently renamed to *LiveContent Create*, which we included in the software analysis, is a Java-based program for embedding in web pages. Uniquely (so far as we have been able to ascertain) it implements both Target Markup Adoption and the ability to map the invocation of italics and bold to a choice of element types.

The Wikimedia editor is an ongoing experiment to develop a What You See Is What You Meant (WYSIWYM) in-browser editor. Implementations include a two-panel editor, with synchronous typographic editing on one side and a wiki markup editor on the other, with real-time (synchronised) shadowing between them. If you edit in one pane, the edits happen as you type in the other. While this does not yet implement any of our proposed interface changes, it does seem to show that real-time editing can be achieved synchronously between a typographic display and a markup display.

A technical support site for TEX, `http://tex.stackexchange.com`, has near-real-time translation of TEX source code to graphical display in the browser; and the easily-installed *MathJax* web server rendering library means that any web site can offer TEX-style mathematical formatting. Although we did not investigate mathematics editing as part of this research, the synchronous nature of the solution also indicates that some of the problems of real-time interpretation of markup can be overcome.

An even more recent introduction (April 2013) is *Fidus Writer*, a web application for editing structured documents for export to LaTeX as well as PDF and EPUB formats. While the current implementation is still under development, the conceptual approach is very close to the paradigms we outlined earlier: a WYSIWYG interface, no markup on view, and adherence to the perceived user expectations of keystroke and menu behaviour.

Among the conclusions of the 2013 XML Summer School was that HTML5 was likely to become the editing file format of choice in the publishing industry, largely because of its widespread base as the file format in EPUBs, a view raised a month earlier in 'The Case for Authoring and Producing Books in (X)HTML5' (Kleinfeld, 2013). If this is the case, the next few years should see a significant increase in HTML5 editors, and in existing editors offering to save documents in HTML5 or EPUB format (some already do).

However, publishers do not use *Word* because they love its interface; they use it because of its perceived prevalence on authors' desktop computers (even when it's not), and for one overwhelming reason: Change Tracking (Reviewing). *Word*'s implementation of this feature allows publishers' editors to exchange drafts with authors in a consistent if complex manner that is unavailable to such depth in LaTeX and native-XML editors without significant specialist intervention. While many publishers have invested heavily in XSLT transformations from *Word* to other formats in the publishing pipeline, it remains true that the OOXML file format is cumbersome in the extreme, and grotesquely inefficient for these purposes, whatever its advantages to Microsoft for re-editability. HTML5 is not a panacæa, but there can be no doubt that it would provide a vastly simpler and more robust format for editing, whether or not the interface includes any of the modifications suggested here.

As if in parallel with this, two other solutions also offer alternative approaches.

The *Markdown* format (`http://daringfireball.net/projects/markdown/`) has been popular since its inception in 2004. It is a codification of many existing habits of plaintext email users (asterisks surrounding bold, underscores surrounding italics, flag characters for headings, etc), plus a program that converts this to HTML. The result is that an author can write with very little effort in a simple structure, but remain confident that the result will at least work well in a web page (for example, pasted into a content management system). While it offers little sophistication, it can easily be applied to the very large class of documents such as web pages or novels, which only need an extremely light

markup structure. There is a growing collection of systems online and offline which use the format, some with an interactive preview and multiple export formats. There is however a question mark over the future of web-based collaborative writing: one of the most promising, *Editorially* (`https://editorially.com/`), had to close down due to lack of support.

Poetica (`https://poetica.com/`) is a nascent system for authoring and editing, but specifically for allowing changes to be signalled in a manner much closer to the pen-and-paper method, but preserving the document structure and a HTML5 file format. These two key features tackle head-on the very reasons behind publishers' current adoption of *Word*.

Scrivener is altogether a different concept: more like the specialist software referred to in varlistentry section 1.1.3.1, para Chapter 2, and para section 2.2.2. It handles structural and conceptual documents with relative ease from a literary (writers') point of view, and retains the formatting-oriented toolbar. It also exports to *Word*, LaTeX, and other formats. The interface is relatively complex, perhaps as a result of attempting to please all of the people all of the time, but it is very popular with literary authors, especially on Mac OS X where it was first developed. While it certainly deals with document structure, its handling is less rigorous than would be needed for authoring to a predefined structure.

Ultimately, the choice will not lie with the author, despite the evident actual and potential use of the HTML5 format as a *lingua franca*. There is still too much jockeying for position among vendors, each one with marketing teams anxious to demonstrate that they can create a Unique Selling Proposition that will invalidate their competitors' claims and lock in customers to their product for the foreseeable future. Publishers may yet have the chance to be the driving force in software choice.

6.4.3 Further work

The development of markup and editing technology has given us syntaxes and toolsets with which we can express much greater generality about document structure. Vesalius had to struggle to convey his instructions to Oporinus. Later generations had to learn how to edit and mark up text for typesetting (as in this paragraph) so that they could take advantage of that era's technology. The wordprocessor user continues to struggle with the tedious and repetitive task of highlighting text and clicking on font style and size dropdowns, only to have to go

back and change it all whenever it is edited.

The markup user puts the cursor in the text and clicks Title in the stylebar, and the job is done.

In Chapter 1 we presented evidence that humans have been considering structure in documents for a very long time, but without automation, individuals could directly affect only their own work. The changes we have discussed here have enabled a great deal of thinking about document structure and design by semanticists, document type designers, wordprocessor and other software designers, graphic designers, programmers, editors, and many others with related concerns — including authors themselves. This has allowed large numbers of people to actively *do* something about it in their own work or that of others, and thereby introduce much greater efficiency and effectiveness.

As we saw in Chapter 4, the 'ribbon' control interface introduced in *Word* has not yet been widely adopted in other Windows software. Large software manufacturers are naturally conservative and slow to adopt radical new departures of their own accord, LyX preferring to let smaller, more adaptable companies or individuals demonstrate the viability first. Despite commercial antagonism to FLOSS, some of this attitude must encourage its growth, where entire new products can be written by solo programmers or small, widely-dispersed teams, able to react swiftly to new paradigms. In reality, companies may be doing themselves and their shareholders a disservice by not making the first move.

The results of this research show that there is much scope for further work in this area. The relatively small number of functions selected for testing was restricted purely on grounds of practicality: similar criteria could well be applied to all the other menu and toolbar items in the generic interface. We have already cited the LyX editor for LaTeX as having dispensed entirely with the font name, style, and size dropdowns (see Figure 4.10 on page 256), on the grounds that the stylesheet (document class or packages) provides all the facilities needed via the semantic controls. This prescriptive view will not be applicable to all users, or to those wanting to 'control' their document directly, but as we have argued in section 6.3.12 on page 357, there is no reason why modifications carried out via the font dropdowns should not be adopted into the current stylesheet.

The 'Web 2.0' interfaces popularised by social media sites and portable devices introduced a new way of working, where everything becomes an affordance.

Long-press on text, and it becomes editable; on an image, and it becomes shareable. In a restricted space, with little room for menus and buttons, changes to settings take immediate effect, and the Back button merely exits the Settings mode. Some of this has already made its way into new laptop and desktop interfaces such as Ubuntu's *Unity* and *Touch*, and Microsoft's *Windows 8* (Mac OS X *Mavericks* appears to be trailing despite the iPhone having been a standard-bearer for the new paradigms). While there are obvious physical limitations to small mobile devices when it comes to writing and editing text documents, voice recognition and the ubiquity of pocket Bluetooth keyboards mean that it is by no means impossible, and the new interfaces may be able to do away with menus and toolbars entirely.

Practical recognition of document structure based on typographic variation has always been attractive, and systems are still being developed to extract such metadata from PDF and even Printer Command Language (PCL) print streams. For a large class of canonical document structures, however, systems like *Markdown*, the Wikimedia visual editor, *Editorially*, or *Poetica*, producing HTML5 seem likely to lead the next phase of development, and will require many of the aspects of the interface mentioned above.

The reusability of documents has always been a cornerstone of the generic markup philosophy. Making the markup generic was the first phase, and creating a usable interface was the second (leaving aside the problems of parser-writing and stylesheet design). With the rich toolsets now available, even to *Word* users, there seems little excuse outside specialist areas for deliberately creating documents that *cannot* be passed seamlessly between otherwise disparate systems. An engineering colleague summed it up:

> Pick the right tools and re-use is comparatively easy. Pick the wrong tools, and all the work of preparation becomes unique to each edition and medium.

Appendix A

Expert survey

The page reproductions of the questionnaire start overleaf.

Editing software for structured document applications:
Expert questionnaire (administered)

There are 7 questions I would like to discuss with you about your use of editors for structured-document applications. All of them are optional: please just say if you prefer not to answer one. The questionnaire is anonymous and the data will be used only for the purposes of the administrator's research.

1 Background

To start, please let me know something about your background in using structured-document systems.

Occupation .....
Affiliation[s] ☐ Private ☐ Corp ☐ Univ ☐ W3C ☐ OASIS ☐ TUG ☐ IdeAlliance
Operating system experience ☐ DOS ☐ Win ☐ Unix ☐ Mac ☐ Mini ☐ MF ☐ Other .....
Structured document experience ☐ Dot ☐ Script/GML ☐ LaTeX ☐ SGML ☐ XML ☐ Other .....
Years using structured documents ☐ 1–2 ☐ 3–5 ☐ 6–10 ☐ 11–15 ☐ 15+
Application types ☐ Document ☐ Data ☐ E-commerce ☐ Repository ☐ Other .....

2 Selection criteria

What are the criteria you use to decide what editor is best suited to a given project?

.....
.....
.....
.....
.....

What would be your personal *choice of product for the following tasks (assuming XML, but otherwise you have a completely free hand to choose, ie unconstrained by cost, platform, or anything else)?*

	Writing it yourself	Editing others' writing
A newpaper or magazine article on the popularity of PDAs	.....	.....
The Owner's Manual for a washing machine	.....	.....
A romantic novel (eg Mills & Boon)	.....	.....
Documentation of your own code	.....	.....
Your memoirs	.....	.....
Maintenance documentation for a ski-lift	.....	.....
An article on Perl for a conference or journal	.....	.....
A picture-and-story book for ages 4–8	.....	.....
A book on the architecture of Oxford	.....	.....
The leaflets included in consumer electronics purchases	.....	.....

3 Are there products you would prefer to use but cannot (for reasons of price, licensing, availability, platforms, etc)?

.....
.....
.....
.....
.....

Over...

4 Can you identify up to three things your preferred software does which you consider marks it out as specially useful?

Please indicate if they are Features (something the vendor believes is special), Functions (facilities required in order to operate), or Behaviours (ways of doing things).

i) .....

..... ☐

ii) .....

..... ☐

iii) .....

..... ☐

5 Can you identify up to three things about any of the structured-document software that you have tested or used which you consider is particularly poor?

Please indicate if you believe these to be Bugs, Deliberate but flawed design choices, or Carelessness.

i) .....

..... ☐

ii) .....

..... ☐

iii) .....

..... ☐

6 Are there any other facilities in which structured-document software you have experience of is specially lacking?

.....

.....

.....

.....

.....

.....

7 While using a structured-document software product, have you ever failed to identify how to do something the product was actually capable of?

Please include any occasions where you did eventually find out how to do it, even if it took a long time. Can you attribute this to failures in Documentation, Support, or Design?

.....

..... ☐

.....

..... ☐

.....

..... ☐

Thank you very much. Please email `peter.flynn@mars.ucc.ie` *if you would like a copy of the finished research.*

Appendix B

Requests analysis

Main programs used to retrieve Usenet posts from Google Groups.

B.1 Shell script to do a search

```
#! /bin/bash

# Bourne shell script to perform a recursive retrieval on Google
# Groups to find the ORIGINAL post for any thread containing posts
# which contain the specified search string.

##############################################################################
#
# Dependencies
#
# In addition, the conventional suite of Unix text tools (awk, sed,
# grep, etc) is required.

WGET=`which wget`
TIDY=`which tidy`
JAVA=`which java`
SAXON=/usr/local/saxon/b9/saxon9.jar

##############################################################################
#
# Search parameters
#

SITE=groups.google.com
GROUP="$1"
if [ "$GROUP" = "ctt" ]; then
    GROUP=comp.text.tex
elif [ "$GROUP" = "ctx" ]; then
    GROUP=comp.text.xml
else
    echo Group must be the first argument
    exit 1
fi
BASE=~/Personal/phd/usenet
FROM=2006-01-01
NOW=`date -I`
```

```
# A Google search done with the Advanced Search page has the following
# format of URI:
#
# http://groups.google.com/groups/search?as_q=foo&as_epq=&as_oq=&as_eq=&
#    num=10&scoring=&lr=&sitesearch=groups.google.com&as_qdr=&as_drrb=b&
#    as_mind=1&as_minm=1&as_miny=2006&as_maxd=1&as_maxm=1&as_maxy=2009&
#    as_ugroup=comp.text.tex&as_usubject=&as_uauthors=&safe=off
#
# The next n are returned with
#
# http://groups.google.com/groups/search?as_q=foo&as_epq=&as_oq=&as_eq=&
#    num=10&scoring=&lr=&sitesearch=groups.google.com&as_qdr=&as_drrb=b&
#    as_mind=1&as_minm=1&as_miny=2006&as_maxd=1&as_maxm=1&as_maxy=2009&
#    as_ugroup=comp.text.tex&as_usubject=&as_uauthors=&safe=off&sa=N&
#    start=10

##############################################################################
#
# Must have arguments to search for, passed from the command line

if [ -n "$1" ]; then

# Catenate the terms, get rid of leading, trailing, and multiple spaces
# turn spaces into + signs (URLencoding)

    TERMS=‘echo $2 $3 $4 $5 $6 $7 $8 $9 |\
           sed -e "s+^[\ ]*++" -e "s+[\ ]*$++" -e "s’\ ’+’g"‘

##############################################################################
#
# Change to the working directory, create a subdirectory for this group
# if there isn’t one already, and change into it

cd $BASE
mkdir -p $GROUP
cd $GROUP

##############################################################################
#
# Remove all spaces from the search terms to create a string label for
# this search, used as the subdirectory name. Delete any previous
# subdir by this name for this target

    SEARCH=‘echo $2 $3 $4 $5 $6 $7 $8 $9 | sed -e "s’\ ’’g"‘
    /bin/rm -rf $SEARCH
    mkdir -p $SEARCH

##############################################################################
#
# Reset the hit counters and the increments. Split up the dates.

    START=0
    MAX=$[START+9]
    THREADS=0
    MORE=yes
    FYR=‘echo $FROM|awk -F- ’{print $1}’‘
    FMN=‘echo $FROM|awk -F- ’{print gensub("^0","",1,$2)}’‘
    FDY=‘echo $FROM|awk -F- ’{print gensub("^0","",1,$3)}’‘
    TYR=‘echo $NOW|awk -F- ’{print $1}’‘
    TMN=‘echo $NOW|awk -F- ’{print gensub("^0","",1,$2)}’‘
    TDY=‘echo $NOW|awk -F- ’{print gensub("^0","",1,$3)}’‘

    echo Searching for $TERMS from $FDY/$FMN/$FYR to $TDY/$TMN/$TYR

##############################################################################
#
# Start the search loop, 10 threads at time

    while [ -n "$MORE" ]; do
```

```
        LIMIT=$[MAX-START+1]

        echo
        echo Batch $START to $MAX
        echo

# wget the search from Google Groups into the search file

        echo Getting listing
        $WGET -U "Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8)" \
            -O $SEARCH/get$START.html \
            "http://groups.google.com/groups/search?as_q=$TERMS&as_epq=&
as_oq=&as_eq=&num=$LIMIT&scoring=&lr=&sitesearch=$SITE&as_qdr=&as_drrb=b&
as_mind=$FDY&as_minm=$FMN&as_miny=$FYR&as_maxd=$TDY&as_maxm=$TMN&
as_maxy=$TYR&as_ugroup=$GROUP&as_usubject=&as_uauthors=&safe=off&
start=$START" \
            2>$SEARCH/get$START.log

# This returns a HTML document get<n>.html with a link for each thread thus:
#
# http://www.google.com/url?url=http://groups.google.com/g/7b67feea/t/
# 139de4a2e05b71c2/d/d9483c29730a384b%3Fq%3Deasy%2Bto%2Buse%2Bxml%2Beditor
# %2Bgroup:comp.text.xml%23d9483c29730a384b&ei=3NTLSbKyAaG4Q8frvKQL&sa=t&
# ct=res&cd=1&source=groups&usg=AFQjCNGzOHOiUK6IxvRTOtuIjksvEbxNDg

##############################################################################
#
# Tidy the file into XHTML

        echo Tidying listing
        $TIDY -ni -asxml -f $SEARCH/get$START-tidy.err \
                            $SEARCH/get$START.html |\
            sed -e "s+ xmlns=\"http://www.w3.org/1999/xhtml\"++" \
            >$SEARCH/get$START.xml

##############################################################################
#
# Extract each thread link (see getpost.xsl for details) and create
# a script (getthreads) to retrieve them

        echo Extracting threads
        $JAVA -jar $SAXON -o $SEARCH/getthreads$START \
              $SEARCH/get$START.xml $BASE/getpost.xsl \
              start=$START site=$SITE job=$SEARCH threads=$THREADS \
              2>$SEARCH/get$START.err

##############################################################################
#
# Execute that script to get the threads. This does the message retrieval

        . $SEARCH/getthreads$START

##############################################################################
#
# Count how many retrievals were in it, add them to the counter

        NEWTHREADS=`cat $SEARCH/getthreads$START | grep WGET | wc -l`
        THREADS=$[THREADS+NEWTHREADS]

# see if the Google page had a link to more (used as a token for this loop)

        MORE=`grep 'Next</a></td>' $SEARCH/get$START.xml`

# increment the block counter

        START=$[START+LIMIT]
        MAX=$[START+9]

# and go back for more
```

```
    done

else

    echo Need some search terms

fi

exit 0

# Check
# for m in *.msg; do DATE=`grep '^Date: ' $m | head -1 |\
#    sed -e "s+^Date:\ ++"`; date -d "$DATE" --iso-8601; done
```

B.2 XSLT script to extract the thread link

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

  <!--

       XSLT script to identify thread headers in a Google Groups search
       and write a shell script to get them from Google.

       Executed from within the "search" shell script.

       NOTE: the posts identified by this script are not necessarily
       the OP's original that started the thread, but the post within
       that thread that the Google search engine found. For this reason
       we retrieve the entire thread and then run another XSLT task
       (getsource.xsl) which looks only at the first post in the thread,
       which *is* the OP's original. It's *that* post that we get the
       text from.

       -->

  <xsl:output method="text"/>

  <xsl:param name="site"/><!-- eg http://groups.google.com -->
  <xsl:param name="job"/><!-- eg comp.text.xml-easytousexmleditor -->
  <xsl:param name="threads"/><!-- counter from script -->
  <xsl:param name="start"/>

  <!-- we want all those posts which are contained in a div class="g".
       This used to be those displayed indented 38em but it changed.
       *provided that* they are NOT preceded by a null column 40px wide
       (which is what indicates that they are a follow-up) -->

  <xsl:template match="/">
    <xsl:apply-templates select="//div[@class='g']"/>
  </xsl:template>

  <!-- write the script, picking the link from the first <a> element -->

  <xsl:template match="div">
    <xsl:text>&#xA;echo Getting thread </xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text>&#xa;mkdir -p $SEARCH/thread</xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text>&#xA;$WGET </xsl:text>
    <xsl:text>-U "Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8)" </xsl:text>
    <xsl:text>-O $SEARCH/thread</xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text>/index.html "</xsl:text>
```

```
    <xsl:value-of select="a[1]/@href"/>
    <xsl:text>" 2>>$SEARCH/thread</xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text>/getthread.log
echo Tidying thread
    $TIDY -ni -asxml -f $SEARCH/thread</xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text>/tidy.err $SEARCH/thread</xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text>/index.html | </xsl:text>
    <xsl:text>sed -e "s+ xmlns=\"http://www.w3.org/1999/xhtml\"++" </xsl:text>
    <xsl:text>>$SEARCH/thread</xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text>/index.xml
echo Extracting post
    $JAVA -jar $SAXON -o $SEARCH/thread</xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text>/getsource $SEARCH/thread</xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text>/index.xml $BASE/getsource.xsl </xsl:text>
    <xsl:text>site=$SITE job=$SEARCH thread=</xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text> 2>>$SEARCH/thread</xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text>/getthread.err
. $SEARCH/thread</xsl:text>
    <xsl:value-of select="$threads + position()"/>
    <xsl:text>/getsource</xsl:text>
    <xsl:text>
echo Got $MSGID
</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Appendix C

User survey

C.1 Questionnaire form

The page reproductions of the questionnaire start overleaf.

Editing software for structured documents

Peter Flynn, Human Factors Research Group, UCC

Page 1 of 3

This is a survey about software for editing structured documents (books, theses, manuals, reports, articles, etc). It forms a part of my PhD research into the usability of structured editors. The survey is anonymous, but you can leave your email address if you would like a copy of the final results. Please contact me if you have any problems [peter.flynn (at) mars.ucc.ie]. If you cannot give an answer to a question, type a dash (-).

Questions marked with a * are required.

SECTION 1. This section asks about your background and experience in computing and structured document systems.

***1.** **What is your occupation or profession?**

***2.** **What kind of organisation do you work for (if any)?**

- ◯ Big business (corporation)
- ◯ Small/medium business (SME)
- ◯ Education (school, college, university)
- ◯ Research (industrial, non-profit, NGO)
- ◯ Government (federal, state, or semi-state bodies)
- ◯ Consultancy
- ◯ Student (undergraduate, postgraduate)
- ◯ Self-employed (sole proprietorship)
- ◯ Freelance (author, editor)
- ◯ Other:

***3.** **What computer system[s] do you use most often?**

- ☐ Microsoft Windows

- [ ] Apple Macintosh OS X
- [ ] Unix (including GNU/Linux, BSD, etc)
- [ ] Other:

***4. What document system[s] do you use most often?**

- [ ] "Dot-line" (eg Runoff/nroff/troff)
- [ ] Script/GML
- [ ] SGML
- [ ] XML
- [ ] Plain TeX / ETeX
- [ ] LaTeX / ConTeXt / Texinfo
- [ ] WordPerfect
- [ ] Microsoft Word
- [ ] OpenOffice
- [ ] HTML (web pages)
- [ ] Wiki
- [ ] Other:

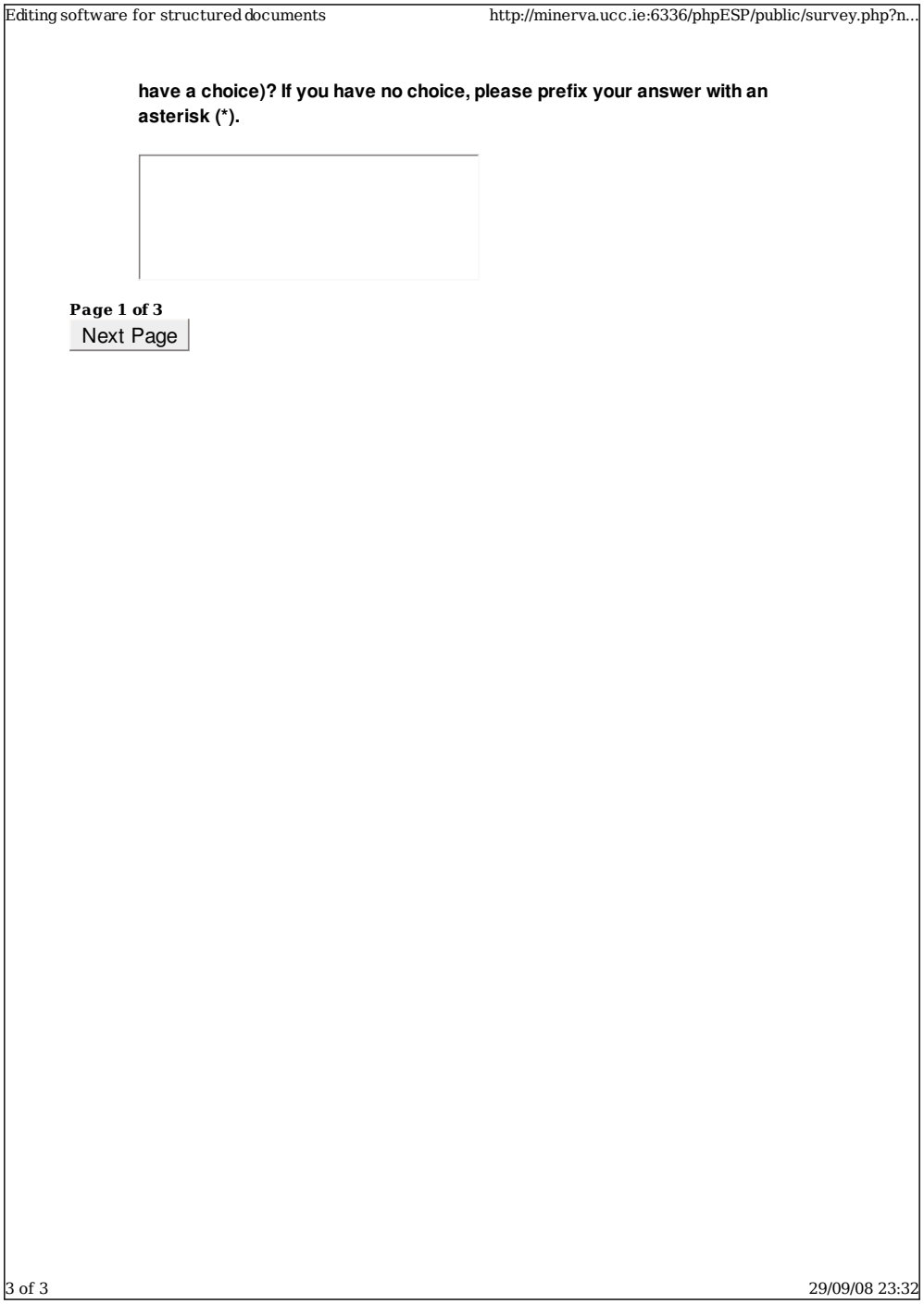
***5. How many years have you been writing documents on a computer?**

- [ ] 0-2
- [ ] 3-5
- [ ] 6-10
- [ ] 11-15
- [ ] 16 and over

***6. What types of documents have you had most experience with?**

- ◯ Text documents (books, reports, articles, theses, essays etc)
- ◯ Structured data (e-commerce, messaging, data interchange, configuration, etc)
- ◯ Web pages (HTML, content management systems, wikis)
- ◯ Multimedia (graphics, audio, video, media synchronisation)
- ◯ Other:

***7. What editing software do you prefer to use for structured documents (if you**


Editing software for structured documents
http://minerva.ucc.ie:6336/phpESP/public/survey.php?n...
have a choice)? If you have no choice, please prefix your answer with an asterisk (*).
Page 1 of 3
Next Page
3 of 3
29/09/08 23:32

Editing software for structured documents

Peter Flynn, Human Factors Research Group, UCC

Page 2 of 3

This is a survey about software for editing structured documents (books, theses, manuals, reports, articles, etc). It forms a part of my PhD research into the usability of structured editors. The survey is anonymous, but you can leave your email address if you would like a copy of the final results. Please contact me if you have any problems [peter.flynn (at) mars.ucc.ie]. If you cannot give an answer to a question, type a dash (-).

Questions marked with a * are required.

SECTION 2. This section asks about how you use your editing software for writing or editing structured documents. The choices are not exclusive: please check all those that apply.

***8.** **How do you create a new document of the right type?**

- ☐ There is a 'New Document' menu where I choose the document type I want
- ☐ I open an existing document of the right type, and delete the old text
- ☐ I manually edit an empty file to specify the type of document
- ☐ There is a document type creator for new types of document I haven't used before
- ☐ The 'New Document' menu doesn't have the document types I want to use
- ☐ There is no choice: I use the File|New menu and just start typing
- ☐ There are sample templates I can use or modify to get what I want
- ☐ I check a skeleton document out of the repository
- ☐ Other:

***9.** **How do you give the title, author, and other key information for a new document?**

- ☐ My editing software asks me for all this as I open a new document
- ☐ There are spaces to fill this in after the new document opens
- ☐ I type it in and format it to look right
- ☐ I type it in with instructions to identify it

- [ ] I have to specify this before creating the new document
- [ ] There is a separate procedure to go through for each document type
- [ ] I don't: it's determined beforehand and inserted automatically
- [ ] Other:

***10. How do you tell your editor to start a new section (or chapter, subsection, subsubsection, etc)?**

- [ ] I click on 'section' (or whatever) in the 'New' menu
- [ ] I move past the end of an existing section and click 'Insert'
- [ ] I position the cursor to the right place and type the sectioning instruction
- [ ] There is no way to do this: I have to type the heading and format it by hand with font and size menus
- [ ] I type the heading and use the template style menu to specify the type of heading
- [ ] I type the heading in a dialog box and label it as a section
- [ ] I position the cursor inside the end of the previous section and split it to make a new one
- [ ] Other:

***11. How do you apply formatting or styling?**

- [ ] I don't need to: my editing software does it for me based on the template style
- [ ] I don't have to: the editorial production team does all that later
- [ ] I use the font menu, the size menu, and the B, I, and U buttons (or their keyboard shortcuts)
- [ ] I highlight the text and use the predefined Styles menu
- [ ] There are toolbar buttons for identifying stuff according to meaning which do the formatting automatically
- [ ] Other:

***12. How do you move blocks of text around when you edit a document? ('Blocks' here means whole paragraphs, items, lists, tables, figures, sections, etc, right up to entire chapters; not words or phrases within a block).**

- [ ] I highlight the text with the cursor, then cut and paste
- [ ] I mark blocks using a structure window, then cut and paste
- [ ] To move whole blocks I have to move their content first, then delete the empty container

- [ ] I mark blocks in the structure window and then click 'Move' and specify the destination in a dialog box
- [ ] I highlight the text with the cursor, then cut and paste, but I may have to reorganise the structuring instructions manually afterwards
- [ ] Other:

***13. How do you navigate around the document when editing?**

- [ ] There is a structure window where I can click to move to a different place
- [ ] I scroll up or down until I find the part I want to edit
- [ ] I use the search to find the word or phrase I want to edit
- [ ] There are navigation buttons to let me move through the structure
- [ ] I use page thumbnail images to scroll through the document
- [ ] I use keyboard shortcuts to jump to or cycle through the structure
- [ ] Other:

***14. How do you add blocks like tables, figures, lists, sidebars, etc?**

- [ ] I use the 'New' menu to insert them
- [ ] There are toolbar buttons or keyboard shortcuts to do these things
- [ ] I click on the 'Insert' menu and pick the one I need
- [ ] I type instructions to identify what I am inserting
- [ ] I drag and drop an icon from the toolbar into the document body
- [ ] I type the text, then use the template style menu to identify it
- [ ] Other:

***15. How do you create or edit linking items like cross-references, footnotes, bibliographic citations, hyperlinks, acronym references, etc?**

- [ ] There are toolbar buttons or keyboard shortcuts to do these things
- [ ] I type the instructions to identify them
- [ ] I use the menus to insert them and then add the information in a dialog box
- [ ] I can drag and drop references from a reference management application
- [ ] The structure window lets me drag and drop cross-reference points
- [ ] I have to create and identify the target item first, then link to it by one of the above means
- [ ] Other:

***16. How do you know what your document-in-progress will look like?**

- ☐ The on-screen formatting in the editing software is good enough
- ☐ I don't need to: it all gets formatted by the production team later
- ☐ I don't worry about it while I'm writing: so long as the structure is right, the formatting will follow, and can be tweaked later
- ☐ There is a typeset preview window showing the updated formatted output
- ☐ There is a toolbar button or keyboard shortcut to display the formatted output
- ☐ I click on Reload in a browser window
- ☐ It's not necessary at the writing/editing stage: the document will be in multiple different formats
- ☐ Other:

Page 2 of 3

Next Page

Editing software for structured documents

Peter Flynn, Human Factors Research Group, UCC

Page 3 of 3

This is a survey about software for editing structured documents (books, theses, manuals, reports, articles, etc). It forms a part of my PhD research into the usability of structured editors. The survey is anonymous, but you can leave your email address if you would like a copy of the final results. Please contact me if you have any problems [peter.flynn (at) mars.ucc.ie]. If you cannot give an answer to a question, type a dash (-).

Questions marked with a * are required.

SECTION 3. This section asks about your experiences with using editing software.

***17.** **What best describes your general approach to creating and maintaining structured documents?**

- ☐ I use editing software that prescribes and enforces structure
- ☐ I use editing software that encourages and supports structure
- ☐ I use editing software that lets me describe and identify the appropriate structure myself
- ☐ I use structuring instructions only where necessary
- ☐ I don't use structured documents
- ☐ Other:

***18.** **When advising others (eg colleagues, clients, students/trainees, friends) about software for editing structured documents, what specific feature[s] do you suggest they look for, or guard against (if any)?**

***19. What is the single most useful feature of the editing software that you use the most? (and what editor is that?)**

***20. What is the single worst feature of any editing software for structured documents that you have used?**

***21. When using an editing system for structured documents, have you ever failed to find out how to do something that the product was actually capable of doing? Please describe the circumstances briefly.**

***22. What features would you like to see in an editing system for structured documents that aren't in any of them at the moment? (ie your WishList)**

23. **If you would like a copy of the results, please enter your email address here.**

24. **Please use this space for comments and feedback about the questionnaire itself.**

Thank you very much for filling in this survey. Addresses will be detached automatically from the survey data to preserve anonymity, and will only be used for sending results. UCC is a strict adherent to the provisions of the Data Protection Acts 1988 and 2003: personal data will never be provided to any third party, and will be deleted when the results have been sent.

Page 3 of 3

Submit Survey

C.2 Routines used in data processing

Programming for data acquisition and control was done largely in *bash*(1) shell scripts, with some in *awk*(1) for data manipulation. The statistical processing was done using the P-Stat package (Buhler & Buhler, 1990).

C.2.1 The `makedata.awk` script

```
# awk(1) script to extract coded column data from TAB-separated file
# of survey data exported from oocalc, and write results as a P-Stat
# transfer file to build the dataset, omitting the original survey
# field data, which is all textual: we only want the encoded data.
#
# Peter Flynn, October 2008
#
# The data format is:
#
# row 1: first tokens of the variable names (encoded columns only)
# row 2: second tokens of the encoded variable names and first tokens
#        of the original survey field names (uppercase)
# row 3: the encoding letters used as a reference while encoding
# row 4 onwards: data
#
# The variable names of the encoded columns (all lowercase) need to
# be concatenated with a period to form the full names, eg for the
# Operating System (OS) question, which had three possible responses
# (Windows, Mac, Unix), the top of the data looks like:
#
# row 1:                    os  os  os
# row 2: OS                 win mac unix
# row 3:                    W   M   U
# row 4: Microsoft Windows  1   0   0
#
# This results in four variables, OS, os.win, os.mac, and os.unix
# with data OS="Microsoft Windows", os.win=1, os.mac=0, os.unix=0
#
# Exception: question 5 ("How many years have you been writing
# documents on a computer?") was a numeric response, so it has been
# retained uncoded (it gets recoded in P-Stat later).

######################################################################
#
# Set the output separator and line-end to null, as we'll be doing
# our own formatting. Start by outputting the P-Stat "build" command
# and options.

BEGIN {
    OFS="";
    ORS="";
    print "build authed, end.of.case '/', vars\n";
}

######################################################################
#
# If this is the first row (first tokens of manually-added variable
# names, all lowercase; and no data in the positions of the original
# survey field names), record the values in array f. Omit any values
# which are numeric: they are the question numbers.

{
    if(NR==1)
```

```
    {
        for(i=1;i<=NF;++i)
        {
            if(gensub("[0123456789]","","G",$i)!="")f[i]=$i;
        }
    }
}

#######################################################################
#
# For row 2, if the ith element of array f is not empty (ie it is
# occupied by a value from row 1) AND the ith value from this row
# is also non-null, catenate the values into the element of array f.
# Treat "exp" as a special case (see above).
# Additionally, if the ith element of array f is not empty, increment
# the line length variable by the length of the field name now in
# the ith element of array f, plus one, and emit a newline if this
# value is greater than 79 (and zero the line length variable). In
# both cases, end by outputting the completed variable name and a
# space. Note that the original survey field names are NOT recorded
# by this method, as we want to use the location of their position
# in the data specifically to omit their data from the output, as
# it is textual, and we only want the encoded data.

{
    if(NR==2)
    {
        linlen=0;
        for(i=1;i<=NF;++i)
        {
            if(f[i]!=""&&$i!="")
            {
                f[i]=f[i] "." $i;
            }
            if($i=="exp") {
                f[i]=$i;
            }
            if(f[i]!="")
            {
                linlen=linlen+length(f[i])+1;
                if(linlen>79)
                {
                    print "\n";
                    linlen=length(f[i])+1;
                }
                print f[i] " ";
            }
        }
    }
}

#######################################################################
#
# At row 3, emit a newline to terminate the subcommand containing
# the fieldnames done at row 2. Ignore the encoding letter data
# and question texts.

{
    if(NR==3)
    {
        print ";\n";
    }
}

#######################################################################
#
# For row 4 onwards (data), if the fieldname in the array f is
# non-null, this signals a column for which data is wanted, so
# measure its length plus one, increment the line length counter
# as for row 2, output a newline if needed, and then output the
```

```
# value.

{
    if(NR>3)
    {
        linlen=0;
        for(i=1;i<=NF;++i)
        {
            if(f[i]!="")
            {
                linlen=linlen+length($i)+1;
                if(linlen>79)
                {
                    print "\n";
                    linlen=length($i)+1;
                }
                print $i " ";
            }
        }
        print "/\n";
    }
}

######################################################################
#
# Finally, terminate the P-Stat command.

END {
    print "$\n";
}
```

C.2.2 The `readdata.xfr` control file

This shows the control command and the first record of data only.

```
build authed, end.of.case '/', vars
resp occ org os.win os.mac os.unix markup.sgml markup.xml markup.latex
markup.wp markup.word markup.oo markup.html markup.goog markup.wiki
markup.marc markup.nb markup.other exp doctype.text doctype.data
doctype.web doctype.mm doctype.other ed.ar ed.bb ed.bx ed.dw ed.em
ed.ex ed.je ed.ki ed.la ed.nb ed.nt ed.oo ed.op ed.ox ed.sw ed.ta
ed.ts ed.te ed.tp ed.tw ed.ue ed.vi ed.wk ed.we ed.wd ed.bp ed.xs
ed.xm create.newdoc create.reuse create.empty create.comp create.nodoc
create.filenew create.sample create.skel meta.ask meta.spaces meta.man
meta.mark meta.prior meta.proc meta.auto meta.oth sect.menu
sect.insert sect.type sect.man sect.temp sect.label sect.split
sect.oth style.auto style.team style.menu style.style style.button
style.oth edit.cursor edit.window edit.cont edit.move edit.reorg
edit.oth nav.window nav.scroll nav.search nav.button nav.thumb
nav.keyboard nav.oth struct.menu struct.button struct.insert
struct.type struct.icon struct.style struct.oth link.button link.type
link.menu link.refman link.dragdrop link.target link.oth view.screen
view.team view.struct view.preview view.button view.reload view.multi
view.oth method.prescribe method.encourage method.self
method.necessary method.nostruct method.oth sel.no sel.oxy sel.comply
sel.nowx sel.keyb sel.nowp sel.cust sel.supp sel.int sel.wx sel.sep
sel.simp sel.goal sel.learn sel.regexp sel.macro sel.complex
sel.mixcon sel.repstr sel.docsiz sel.setup sel.docu sel.noaio sel.nav
sel.tex sel.struct sel.undo sel.unicode sel.choice sel.editauth
sel.editorial sel.hilite sel.optional sel.stable sel.qual sel.intuit
sel.math sel.tabfig sel.xref sel.platform sel.speed useful.autocomp
useful.colour useful.price useful.regexp useful.platform useful.myway
useful.integrate useful.help useful.interfere useful.strmenu
useful.context useful.wysiwyg useful.macro useful.editorial
useful.format useful.features useful.refactor useful.keyb
useful.struct useful.unicode useful.valid useful.wrapping
```

```
useful.schema useful.docu useful.docsize useful.learn useful.stab
useful.xref useful.synch useful.math useful.files useful.undo ;

4205 1 3 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 4 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 /
```

C.2.3 The `reorg.xfr` control file

```
mod authed
[set occ to recode(occ,6 7=1)]
[set org to recode(org,2 5 6 8 9 11=1,3 4 7=2,0=3)]
[set exp to recode(exp,2 3=1,4=2)]
[gen os=sum(os.win to os.unix)]
[gen markup=sum(markup.sgml to markup.other)]
[gen doctype=sum(doctype.text to doctype.other)]
[gen ed=sum(ed.ar to ed.xm)]
[gen create=sum(create.newdoc to create.skel)]
[gen meta=sum(meta.ask to meta.oth)]
[gen sect=sum(sect.menu to sect.oth)]
[keep resp occ org exp
 os os.win os.mac os.unix
 markup markup.sgml markup.xml markup.latex
        markup.wp markup.word markup.oo markup.goog markup.html
        markup.wiki markup.marc markup.nb markup.other
 doctype doctype.text doctype.data doctype.web doctype.mm doctype.other
 ed ed.ar ed.em ed.ex ed.op ed.ox ed.bp ed.xs ed.xm
    ed.ki ed.la ed.sw ed.ts ed.we
    ed.bb ed.bx ed.je ed.nt ed.te ed.tp ed.tw ed.ue ed.vi
    ed.dw ed.ta ed.wk
    ed.nb ed.oo ed.wd
 create create.newdoc create.skel create.sample create.comp
        create.nodoc create.empty create.reuse create.filenew
 meta meta.ask meta.spaces meta.prior meta.auto
      meta.mark meta.proc meta.man meta.oth
 sect sect.menu sect.insert sect.temp sect.label sect.split
      sect.type sect.man sect.oth
],
out authed.clean$
```

C.2.4 Matching patterns with test data

This is the routine for reading the test and pattern data, with the modification to paste the pattern variables onto duplicate copies of the test variables.

```
/* Procedure to read and analyse test data for editor usability changes
   Peter Flynn, June 2013, Human Factors Research Group, UCC */

make tests,file tests.dat,del ',',vars
Tester Task
K.01:c K.02:c K.03:c K.04:c K.05:c K.06:c K.07:c K.08:c K.09:c K.10:c
$

make patterns,file patterns.dat,del ',',vars
Task Pattern
P.01:c P.02:c P.03:c P.04:c P.05:c P.06:c P.07:c P.08:c P.09:c P.10:c
```

```
$

  /* june 16,2013  peter flynn's file matching problem */
  /* courtesy of roald and shirrell buhler on pstat-l  */
  /* (1) Determine max pattern number in the PATTERNS file. */
  /*     Place it in perm scratch var ##maxpat.             */

gen ##maxpat = 0$
mod patterns ( if pattern gt ##maxpat, set ##maxpat to pattern )$
put ##maxpat $

  /* (2) Take TESTS, make WORK1, which has each case of TESTS */
  /*     repeated ##maxpat times.                             */

mod tests ( repeat ##maxpat ), out work1 $

  /* (3) Use DUPLICATES to take WORK1 and create WORK2. */
  /*     This has a new variable named PATTERN          */
  /*     such that it goes from 1 to ##maxpat within    */
  /*     each set of repeated cases.                    */

duplicates work1 [gen pattern], by tester task, all,
           sequence pattern, out work2 $

  /* (4) Use LOOKUP to take WORK2 and create WORK3, */
  /*     appending vars from PATTERNS, using TASK   */
  /*     and PATTERN for linkage.                   */

lookup work2, table patterns, out work3 $

  /* (5) Step 2 made ##maxpat copies of each tester/task      */
  /*     in the input TESTS file, and step 3 sequenced      */
  /*     each of the copies, 1 thru ##maxpat.               */
  /*     These sequence numbers were used to match          */
  /*     pattern numbers. Not all will have been matched,   */
  /*     unless every task has the same number of patterns. */
  /*     So, drop the ones that had no matching pattern.    */

mod  work3 [ if p.01 missing, delete]
           [keep tester task pattern p.? k.?], out work4 $
```

C.2.5 Dataset creation

This process matches the test variables with the pattern variables to identify the pattern which was followed most closely.

```
mod work4[gen E.01:c=.m1.; gen E.02:c=.m1.; gen E.03:c=.m1.;
          gen E.04:c=.m1.; gen E.05:c=.m1.; gen E.06:c=.m1.;
          gen E.07:c=.m1.; gen E.08:c=.m1.; gen E.09:c=.m1.;
          gen E.10:c=.m1.;
          gen S.01=.m1.; gen S.02=.m1.; gen S.03=.m1.; gen S.04=.m1.;
          gen S.05=.m1.; gen S.06=.m1.; gen S.07=.m1.; gen S.08=.m1.;
          gen S.09=.m1.; gen S.10=.m1. ]
[keep p.? k.? e.? s.? .others.]
[gen #p=1;gen #k=1; gen #m=0]
[do #j using p.01 to p.10;
  if v(#k+10) missing, exitdo;
  if v(#p) missing, increase #m, exitdo;
  if v(#p)=v(#k+10),
  then;
    increase #p;
    increase #k;
  else;
```

```
    increase #m;
    set v(#m+20)=v(#k+10);
    set v(#m+30)=#j;
    increase #k;
 endif;
enddo]
[gen p=#p-1,gen k=#k-1,gen m=#m]
[keep tester task pattern p k m .others.],out diffs $

sort diffs, by tester task m(up),out diffs $

mod diffs[if first(tester,task),retain],out best $

sort best,by task,out best;
```

Appendix D

Tester recruitment and introductory procedure

The page reproductions of the questionnaire start overleaf.

Testers sought for interface changes

I'm looking for volunteers to test some changes to a software interface

As part of my research into the usability of software for editing structured documents, I am testing some changes to the way editors and wordprocessors work.

There are 12 changes, and I am looking for volunteers to test the changes to see if they would make the interface more usable. The tests only take a few minutes each, so the whole session should not last more than about 45 minutes.

If you would like to volunteer for a session, please complete this form. For experimental reasons, a random selection of volunteers will be made, so I will email you when I know whether or not you have been selected.

All information here is treated in strict confidence and will be deleted once the random selection is made.

Each of the tests is done once with *Word* or *Libre Office* (your choice) and once with the experimental interface. The whole session should not last more than about 45 minutes.

The sessions will be held at the Usability Laboratory of the Human Factors Research Group in the School of Applied Psychology, North Mall Enterprise Centre (Distillery Fields), UCC. The timing is by arrangement, but will be during June, 2013.

There are 6 questions in this survey

Personal information

Please let us know about your background, work area, and professional skills

[]What is your primary work area? *

Please choose **only one** of the following:

- ○ Academic staff
- ○ Research staff
- ○ Support staff
- ○ Technical staff
- ○ Postgraduate student
- ○ Other

[]Do you write or edit any of the following types of document at work or elsewhere? *

Please choose **all** that apply:

- ☐ Textbooks, business books, technical books
- ☐ Prose fiction
- ☐ Prose non-fiction
- ☐ Articles, white papers, conference presentations
- ☐ Plays, poems, short stories
- ☐ Reports, summaries, minutes, legislation
- ☐ Theses, dissertations, essays
- ☐ None of the above
- ☐ Other: [ ]

[]In total, how many of those documents do you write or edit per year (approximately)? *

Please write your answer here:

[ ]

[]What software do you normally use to write these documents with? *

Please choose **all** that apply:

- ☐ Microsoft Word
- ☐ OpenOffice / Libre Office / NeoOffice
- ☐ Google Docs
- ☐ Some other wordprocessor (specify in Other)
- ☐ LaTeX (specify editor in Other)
- ☐ XML (specify editor in Other)
- ☐ Other: [ ]

[]What fields do most of these documents come under? *

Please choose **all** that apply:

- ☐ Business, Finance, Law
- ☐ Computing
- ☐ Engineering
- ☐ Medical
- ☐ Natural Sciences
- ☐ Social Sciences
- ☐ Humanities
- ☐ Other:

[]How can we contact you? *

Please choose all that apply and provide a comment:

- ☐ Email
- ☐ Phone
- Other:

Thank you for offering to volunteer for the software testing session.

We will be drawing a random selection from among all those who volunteered, within the next two weeks, and will be in touch with you to let you know if your name has been selected or not.

Submit your survey.
Thank you for completing this survey.

D.1 Explanatory notes read to testers

The page reproductions start overleaf.

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

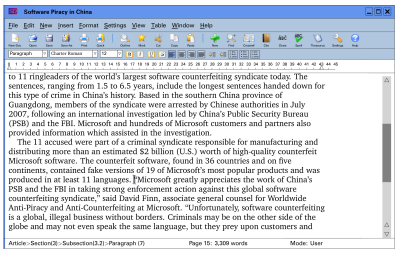
Testing changes to the way we write and edit documents

Please remember that there is no right or wrong in these tests — we are testing the program, not you! The test results are completely anonymous.

This is a research project to test some changes to the way structured document editors and word-processors work.

The program you will be testing is very similar to a normal wordprocessor, but there are a few new icons in the toolbar, and a couple of new menus.

Before we start the test, you will be able to examine the full-size main screen in more detail so that you can see what it looks like.



What's a structured document?

A structured document is one which has to follow a standard pattern, like a business report, a journal article, a thesis, or a book.

A **structured document editor** is like a more powerful wordprocessor which helps you follow the pattern — instead of having to format your document by hand, all the styling is automated.

This means making some changes to the menus and toolbar, and that's what we need to test.

How do you test an editor?

The method we are using to test the program is called **Paper Prototyping**. Rather than write the entire program in advance, only to have to change it again, we use pre-printed screenshots, like the one above.

You can point and click with your finger, or just say what you would click on, and the Experimenter will replace the printout with another one showing the result of your click.

The usability of software for authoring and editing

What are we testing?

We would like you to test twelve tasks. Each of them means making an editorial change to the document, exactly as if you were writing it yourself.

The example document we will use is a magazine article about software piracy, but that isn't really important: it could just as easily be something you would write about in your own field.

Each test takes a few minutes, often less. Just like with a wordprocessor, there are often several different ways to do each task. We are measuring three things:

1. How obvious is it what to click on?
2. How many clicks are needed?
3. Does the program 'do it right' (that is, 'as you expected')?



How does it work?

Each task starts with a short description of what is needed, and the Experimenter will then show you the starting screen.

When you have decided what button to click or when menu to use, point at it, or tell the Experimenter the name of what you would click on.

The Experimenter will then show you the next screen (exactly what you would see if this was a live program).

You can change your mind at any time and go back to the previous screen by telling the Experimenter 'go back' (as this isn't a web browser, there is no Back button).

Is that it?

That's all folks. . . and thank you very much for your time and attention.

Peter Flynn, July 25, 2013

SCRIPT FOR INTRODUCTION

Show tester the 1–page explanation and answer any questions

SAY I'm now going to show you the main editing window that we need to test.

Show basic screen

SAY This is what the editing window looks like. You can see it's in the middle of editing a document, Point to cursor and there's the cursor. It's got the traditional menus and toolbars, but there are a few additions: Point them out



OUTLINE Shows an outline of the sections and subsections in the document, for navigation.



MARK Selects the whole current item. Repeat presses select the container (eg section, then chapter, etc).



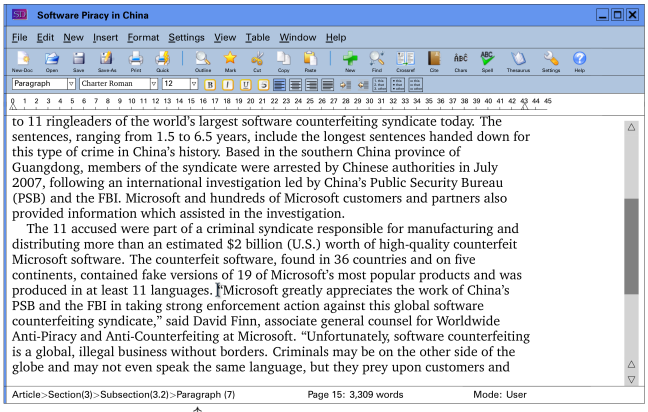
NEW Adds new items such as chapters, sections, paragraphs, lists, figures and tables, in the next place available.



CROSSREF Adds a cross-reference from one place to another in the document.



CITE Adds a citation from your reading list, and add the reference at the end.





***Edit* menu** Contains extra functions for text manipulation



***New* menu** Menu version of the *New* toolbar button



Style menu Pre-set styles for the type of document



List buttons For making or changing lists, including an extra one for description lists

There is a 'breadcrumb' ↑ at the bottom of the screen showing where you are in the document structure.

SAY The document is a magazine article is about software piracy, but it's only an example. It could just as easily be a white paper from your company, a consultancy report, a research article, a book review, or a student essay or thesis — any type of document which follows a well-established pre-set pattern.

We're not interested in the formatting, only in the way the document is put together (the structure). We're trying to see how an editor could be made faster and simpler to use, while still letting you keep to the patterns needed for the document to be re-used in different ways (web, PDF, voice, Braille, etc).

We would like you to test 12 operations using this editor. They are all common things that people do when writing or editing. They don't require any special knowledge or skills, and there isn't any 'right' or 'wrong' involved.

Ask if this makes sense, and answer any questions

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

SCRIPT FOR TESTS

SAY We are now going to start the tests. There are twelve of them, and each one can take anything from a few seconds to a few minutes.

Each test starts with a screenshot of the system editing a document.

For each one, I will explain what stage it is at, and what the next step is, for example "You need to start a new paragraph."

All you need to do is point at the menu or button that you would click on, and I'll make a note of it, and show you the screenshot that would result from it.

If you need to scroll or move the cursor, just say so. In fact feel free to speak about what you are thinking as you do it: this would be very useful to know.

If you want to go back to a previous screen, just say "Back."

When you feel you have done as much as the system lets you, say "Done" and we will end that test.

Remember there is no right or wrong way to do things, only the way you feel is best.

You can stop any test, or the whole session, at any time by saying "Stop test" or "Stop session."

Ask if they understand. Explain anything necessary.

SAY Starting the first test now.

The usability of software for authoring and editing — Peter Flynn, July 25, 2013

D.2 Test recording sheets and scripts

The page reproductions start overleaf.

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 1. Create a new document NEWDOC-START

SAY Let's say you need to write a new document — a journal article. What would you be likely to click on?

IF **New (menu)** ⟶ NEWDOC-NEW ☐

Document... (menu item) ⟶ NEWDOC-NEW-SUBMENU ☐

Article (Journal) (submenu item) ⟶ NEWDOC-SUCCESS ☐

OR **New-Doc (button)** ⟶ NEWDOC-NEW-DOC-BUTTON ☐

Article (Journal) (submenu item) ⟶ NEWDOC-SUCCESS ☐

OR **New (button)** ⟶ NEWDOC-NEW-BUTTON ☐

Article (Journal) (submenu item) ⟶ NEWDOC-SUCCESS ☐

OR **File** ⟶ NEWDOC-FILE ☐

New... (menu item) ⟶ NEWDOC-FILE-SUBMENU ☐

Article (Journal) (submenu item) ⟶ NEWDOC-SUCCESS ☐

ELSE (anything else) write here ☐

SAY OK, that was by way of introduction. Let's go back now to the example article we saw earlier.

The usability of software for authoring and editing Test 1 Peter Flynn, July 25, 2013

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 2. Add a new paragraph after the current one NEWPAR-START

SAY You decide that you need to add a new paragraph, after the one where the cursor is now.

Ask the tester to identify the cursor location

SAY Find the end of the current paragraph, and add a new one immediately after it.

The end of the current paragraph is off the screen, so the tester needs to scroll

IF **Scroll to end of paragraph** ⟶ NEWPAR-CURSOR ..... ☐

IF ***Enter (key)*** ⟶ NEWPAR-SUCCESS ..... ☐

OR ***New (button)*** ⟶ NEWPAR-NEW-BUTTON ..... ☐

Paragraph (menu item) ⟶ NEWPAR-SUCCESS ..... ☐

OR ***New (menu)*** ⟶ NEWPAR-NEW ..... ☐

Paragraph (menu item) ⟶ NEWPAR-SUCCESS ..... ☐

OR **New (button)** ⟶ NEWPAR-NEW-BUTTON ..... ☐

Paragraph (menu item) ⟶ NEWPAR-SUCCESS ..... ☐

OR **Enter (key) 2nd time** ⟶ PARSPLIT-DIALOG ..... ☐

record action ..... ☐

ELSE (anything else) write here ..... ☐

The usability of software for authoring and editing Test 2 Peter Flynn, July 25, 2013

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 3. Split a paragraph into two PARSPLIT-START

SAY You decide that the paragraph *before* the current one (the paragraph you just left) would be better split into two paragraphs, after the sentence ending '...produced in at least 11 languages.' (ie *before* the quoted phrase '"Microsoft greatly appreciates...'.

Allow the tester time to locate this point. Show them if needed.

SAY What would you click on to do this?

IF **Scroll to '"Microsoft greatly appreciates...'** ⟶ PARSPLIT-CURSOR ..... ☐

IF ***Enter (key)*** ⟶ PARSPLIT-SUCCESS ..... ☐

OR ***Edit (menu)*** ⟶ PARSPLIT-EDIT ..... ☐

Split (menu item) ⟶ PARSPLIT-SUCCESS ..... ☐

OR ***Insert (menu)*** ⟶ PARSPLIT-INSERT ..... ☐

Paragraph break (menu item) ⟶ PARSPLIT-SUCCESS ..... ☐

OR **Enter (key) 2nd time** ⟶ PARSPLIT-DIALOG ..... ☐

record action ..... ☐

ELSE (anything else) write here ..... ☐

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 4. Join a paragraph to the preceding one JOINPREC-START

SAY You then decide that the now-separate paragraph beginning 'The 11 accused...' would be better joined to the end of the paragraph before it (the one ending '...assisted in the investigation.')

How would you do this?

Allow the tester time to locate this point. Show them if needed.

IF **Cursor to start of paragraph 'The 11 accused...'** ⟶ JOINPREC-CURSOR ..... ☐

Backspace (key) ⟶ JOINPREC-SUCCESS ..... ☐

OR **Cursor to end of preceding paragraph** ⟶ JOINPREC-CURSOR ..... ☐

Delete (key) ⟶ JOINPREC-SUCCESS ..... ☐

OR **Edit (menu)** ⟶ JOINPREC-EDIT ..... ☐

Join to preceding... (menu item) ⟶ JOINPREC-SUCCESS ..... ☐

OR **Ctrl-J (key)** ⟶ JOINPREC-SUCCESS ..... ☐

ELSE (anything else) write here ..... ☐

The usability of software for authoring and editing Test 4 Peter Flynn, July 25, 2013

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 5. Join a paragraph to the following one JOINFOLL-START

SAY Finally, you settle on joining the paragraph beginning 'Microsoft greatly appreciates…' to the paragraph following it (the one beginning '"Software piracy negatively impacts…').

How would you achieve this?

Allow the tester time to locate this point. Show them if needed.

IF **Cursor to end of paragraph 'wherever they may be."'** ⟶ JOINFOLL-START .. ☐

Delete (key) ⟶ JOINFOLL-SUCCESS ..... ☐

OR **Cursor to start of following paragraph** ⟶ JOINFOLL-START ..... ☐

Backspace (key) ⟶ JOINFOLL-SUCCESS ..... ☐

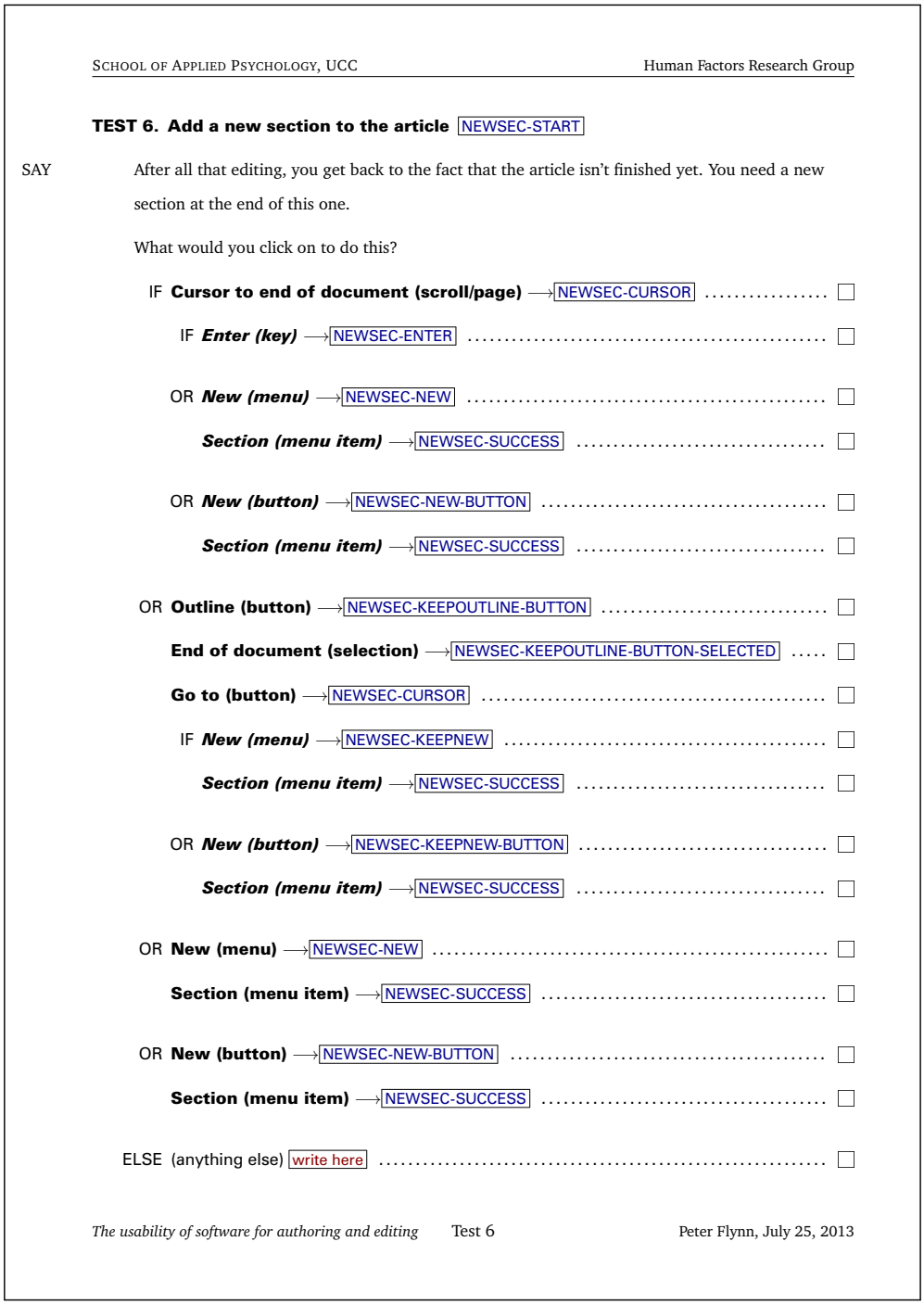
OR **Edit (menu)** ⟶ JOINFOLL-EDIT ..... ☐

Join to following… (menu item) ⟶ JOINFOLL-SUCCESS ..... ☐

OR **Shift-Ctrl-J (key)** ⟶ JOINFOLL-SUCCESS ..... ☐

ELSE (anything else) write here ..... ☐

The usability of software for authoring and editing Test 5 Peter Flynn, July 25, 2013



SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 6. Add a new section to the article NEWSEC-START

SAY After all that editing, you get back to the fact that the article isn't finished yet. You need a new section at the end of this one.

What would you click on to do this?

IF **Cursor to end of document (scroll/page)** ⟶ NEWSEC-CURSOR ☐

IF ***Enter (key)*** ⟶ NEWSEC-ENTER ☐

OR ***New (menu)*** ⟶ NEWSEC-NEW ☐

Section (menu item) ⟶ NEWSEC-SUCCESS ☐

OR ***New (button)*** ⟶ NEWSEC-NEW-BUTTON ☐

Section (menu item) ⟶ NEWSEC-SUCCESS ☐

OR **Outline (button)** ⟶ NEWSEC-KEEPOUTLINE-BUTTON ☐

End of document (selection) ⟶ NEWSEC-KEEPOUTLINE-BUTTON-SELECTED ☐

Go to (button) ⟶ NEWSEC-CURSOR ☐

IF ***New (menu)*** ⟶ NEWSEC-KEEPNEW ☐

Section (menu item) ⟶ NEWSEC-SUCCESS ☐

OR ***New (button)*** ⟶ NEWSEC-KEEPNEW-BUTTON ☐

Section (menu item) ⟶ NEWSEC-SUCCESS ☐

OR **New (menu)** ⟶ NEWSEC-NEW ☐

Section (menu item) ⟶ NEWSEC-SUCCESS ☐

OR **New (button)** ⟶ NEWSEC-NEW-BUTTON ☐

Section (menu item) ⟶ NEWSEC-SUCCESS ☐

ELSE (anything else) write here ☐

The usability of software for authoring and editing Test 6 Peter Flynn, July 25, 2013

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 7. Adding a new list NEWLIST-START

SAY Elsewhere in the article, you feel that you need to list the stages that a piracy investigation goes through, with a numbered list right after the paragraph shown.

What would you click on to add a numbered list at the end of this paragraph (ie between this paragraph and the next one).

Show the tester where, if necessary.

IF **Numbered List (button)** ⟶ NEWLIST-SUCCESS ..... ☐

OR **New (menu)** ⟶ NEWLIST-NEW ..... ☐

Numbered List (menu item) ⟶ NEWLIST-SUCCESS ..... ☐

OR **New (button)** ⟶ NEWLIST-NEW-BUTTON ..... ☐

Numbered List (menu item) ⟶ NEWLIST-SUCCESS ..... ☐

OR **Move to end of paragraph (mouse/arrow)** ⟶ NEWLIST-CURSOR ..... ☐

IF ***Numbered List (button)*** ⟶ NEWLIST-SUCCESS ..... ☐

OR ***New (menu)*** ⟶ NEWLIST-NEW ..... ☐

Numbered List (menu item) ⟶ NEWLIST-SUCCESS ..... ☐

OR ***New (button)*** ⟶ NEWLIST-NEW-BUTTON ..... ☐

Numbered List (menu item) ⟶ NEWLIST-SUCCESS ..... ☐

OR ***Enter (key)*** ⟶ NEWLIST-BLANKLINE ..... ☐

1. ***(digit 1 and dot)*** ⟶ NEWLIST-SUCCESS ..... ☐

ELSE (anything else) write here ..... ☐

The usability of software for authoring and editing Test 7 Peter Flynn, July 25, 2013

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 8. Move a block of text from one place to another [MOVE-START]

SAY You also decide that the text from 'Software piracy negatively impacts. . . ' to the end of the paragraph at '. . . China's knowledge economy.' would go better right at the end of section 3, just before the new section you created in Test 6.

What would you do to move the whole paragraph to that place?

[You can remind the tester about the Mark button.]

IF **Mark (button)** ⟶ [MOVE-MARKED] ☐

OR **Move cursor to start of paragraph (mouse/arrow)** ⟶ [MOVE-CURSOR] ☐

Highlight to end of paragraph (click-mouse/arrow) ⟶ [MOVE-MARKED] ☐

OR **Edit (menu)** ⟶ [MOVE-EDIT] ☐

Mark (menu item) ⟶ [MOVE-MARKED] ☐

ELSE (anything else) [write here] ☐

[Paragraph must be marked by now in order to proceed.]

IF **Outline (button)** ⟶ [MOVE-OUTLINE-BUTTON] ☐

Select 'Customs admin. . . ' (click) ⟶ [MOVE-OUTLINE-BUTTON-SELECTED] ☐

Move marked (button) ⟶ [MOVE-SUCCESS] ☐

OR **Outline (button)** ⟶ [MOVE-OUTLINE-BUTTON-CUT] ☐

Select 'Customs admin. . . ' (click) ⟶ [MOVE-OUTLINE-CUT-BUTTON-SELECTED] ☐

Move marked (button) ⟶ [MOVE-SUCCESS] ☐

OR **Cut (button or Ctrl-X)** ⟶ [MOVE-CUT] ☐

Scroll to before Section 4 (mouse or arrow) ⟶ [MOVE-SCROLL] ☐

Paste (button or Ctrl-V) ⟶ [MOVE-SUCCESS] ☐

ELSE (anything else) [write here] ☐

The usability of software for authoring and editing Test 8 Peter Flynn, July 25, 2013

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 9. Highlighting a trade name (product name) INLINE-START

SAY On your way through the document, you spotted a reference to 'Windows Genuine Advantage'. This is a registered mark of Microsoft Corporation, and your publisher require that all such names must be highlighted. The normal highlight for product names is italics.

What would you do to make this happen?

IF **Highlight text (click/mouse)** ⟶ INLINE-CURSOR ..... ☐

IF ***I*** ***(button)*** ⟶ INLINE-CURSOR-I ..... ☐

Product name (menu item) ⟶ INLINE-SUCCESS ..... ☐

OR ***Ctrl***–***I*** ***(keys)*** ⟶ INLINE-CTRL-I ..... ☐

Product name (menu item) ⟶ INLINE-SUCCESS ..... ☐

OR ***Format (menu)*** ⟶ INLINE-FORMAT ..... ☐

Product name (menu item) ⟶ INLINE-SUCCESS ..... ☐

ELSE (anything else) write here ..... ☐

The usability of software for authoring and editing Test 9 Peter Flynn, July 25, 2013

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 10. Add a cross-reference to another section XREF-START

SAY Nearly done. Having added a lot more to section 3, you need to make sure that readers realise you will be mentioning copyright in section 4, so you want to refer them to it from the end of the paragraph that finishes '. . . China's knowledge economy.'

Show where the cursor is, and how you've started typing 'See'

What would you do to achieve this?

OK to remind the tester of the Crossref button.

IF **Crossref (button)** ⟶ XREF-DIALOG ☐

4. Copyright infringed (menu item) ⟶ XREF-SUCCESS ☐

OR **Insert (menu)** ⟶ XREF-INSERT ☐

Cross-reference (menu item) ⟶ XREF-INSERT-SUBMENU ☐

Insert reference (submenu item) ⟶ XREF-DIALOG ☐

4. Copyright infringed (menu item) ⟶ XREF-SUCCESS ☐

OR **Edit (menu)** ⟶ XREF-EDIT ☐

Find (menu item) ⟶ XREF-EDIT ☐

4. Copyright infringed (menu item) ⟶ XREF-SUCCESS ☐

ELSE (anything else) write here ☐

The usability of software for authoring and editing Test 10 Peter Flynn, July 25, 2013

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 11. Give a reference to an article you read CITE-START

SAY

You have finally tracked down the source of the 'investigations elsewhere' that you said have shown that the pirating of software is not limited to reproducing it in a counterfeit of the original manufacturer's packaging.

It was in a 1999 article by Paul Paradise called 'Trademark counterfeiting, product piracy, and the billion dollar threat to the U.S. economy', and Emphasize you have already added this to your database of references.

After the sentence in section 4 where you mention these investigations, you can now add a citation to Paradise's article.

Show the tester where.

How would you do that?

OK to remind the tester of the Cite button.

IF **Cite (button)** ⟶ CITE-DIALOG ..... ☐

Paradise (selection) ⟶ CITE-SUCCESS ..... ☐

OR **Insert (menu)** ⟶ CITE-INSERT ..... ☐

Citation. . . (menu item) ⟶ CITE-DIALOG ..... ☐

Paradise (selection) ⟶ CITE-SUCCESS ..... ☐

ELSE (anything else) write here ..... ☐

Point out automated addition of References.

The usability of software for authoring and editing Test 11 Peter Flynn, July 25, 2013

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

TEST 12. Insert a fragment of another document INSDOC-START

You realise that the article has to have your name at the very end.

The publisher keeps a standard set of name-blocks in a file called `Author-bylines.xml`, for this very purpose, and you know your name is in there.

What would you click on to insert a piece of another document?

IF Scroll or page to bottom ⟶ INSDOC-CURSOR ..... ☐

Insert (menu) ⟶ INSDOC-INSERT ..... ☐

Document or fragment... ⟶ INSDOC-INSERT-SUBMENU ..... ☐

Your Name (entry) ⟶ INSDOC-SUCCESS ..... ☐

OR Outline (button) ⟶ INSDOC-OUTLINE-BUTTON ..... ☐

End (entry) ⟶ INSDOC-CURSOR ..... ☐

Insert (menu) ⟶ INSDOC-INSERT ..... ☐

Document or fragment... ⟶ INSDOC-INSERT-SUBMENU ..... ☐

Your Name (entry) ⟶ INSDOC-SUCCESS ..... ☐

ELSE (anything else) write here ..... ☐

The usability of software for authoring and editing Test 12 Peter Flynn, July 25, 2013

SCHOOL OF APPLIED PSYCHOLOGY, UCC — Human Factors Research Group

POST-TEST DISCUSSION

SAY That's the last one, thank you very much for your help. If you have a few minutes, I'd be grateful if you could let me have a few reactions to what you have seen and tested.

If they cannot stay

SAY Thanks again.

Otherwise

SAY 1. How obvious was it what to click on?

Very | Mostly | Not much | Not at all ..... ☐

SAY 2. Did you feel it needed more or fewer clicks than your current system?

Ask what current system they use ..... ☐

SAY 3. Do you feel that the program is 'doing it right' (that is, 'as you would expect')?

If not, what would they change? ..... ☐

SAY Thank you again.

The usability of software for authoring and editing Test 12 Peter Flynn, July 25, 2013

Brief background to the tests

This references the section[s] of my thesis where the topic is discussed, or where the original demand for the feature was recorded.

TEST 1. Create a new document NEWDOC

Because of the need to guide document formation with a DTD/Schema or template, creating a new document means that a document type (LaTeX: document class) *must* be selected by some means. Most editors offer a (limited) choice which needs to be *much* wider. It also means that authors should be able to change the formatting via some kind of edit interface not addressed here; possibly similar to *Panorama*'s) without affecting the underlying specification of markup.

☞ Requests Analysis (section 3.3.5, p. 164); User Survey (section 3.4.3.2, p. 183); Modelling & Testing (section 4.3.2, p. 229; Figure 4.9, p. 233).

TEST 2. Add a new paragraph after the current one NEWPAR

The convention of using the Enter key to create a new instance of the current element type is too well-established to overlook, but the key has been used for many other operations in other software, and is ambiguous when the writer wants some other element than the current one (escaping from lists is a common example). We provide for the detection of multiple consecutive presses of the Enter key, and we test a 'New' button/menu which scrolls to the next valid point of insertion for the requested element type before inserting it.

☞ Software Analysis (section 3.2.5.1, p. 134); Modelling & Testing (section 4.3.1.1, p. 214; Figure 4.4, p. 220; section 4.3.3, p. 232)

TEST 3. Split a paragraph into two PARSPLIT

The use of the Enter key for this function is an edge case of the preceding function (where the creation of a new instance is in fact just the splitting of the element into its existing self plus a new, empty one). It is common as an editorial function, but where embedded (inline) markup is present (mixed content) the handling of orphan subelements needs to be dealt with correctly.

☞ Software Analysis (section 3.2.5.1, p. 134; section 3.2.6.2, p. 147); Modelling & Testing (section 4.3.1.1, p. 214)

TEST 4. Join a paragraph to the preceding one JOINPREC

This and the following test are a mirror pair; common in editorial functions, perhaps less so in authoring. As with element-splitting, joining requires its own set of rules to avoid conflict of markup.

☞ Software Analysis (section 3.2.5.1, p. 134); Modelling & Testing (section 4.3.1.4, p. 226; Figure 4.8, p. 227)

TEST 5. Join a paragraph to the following one JOINFOLL

☞ *ibid*.

TEST 6. Add a new section to the article NEWSEC

The conventional scroll–find–move–insert paradigm can be replaced by the 'New' button (see Test 2). This would apply equally to all sectional division levels.

☞ User Survey (section 3.4.3.4, p. 184); Modelling & Testing (section 4.3.3, p. 232; section 4.3.3.1, p. 234)

TEST 7. Adding a new list NEWLIST

Similar to the preceding test, but designed to test if the writer expects spacing around the list to occur automatically (some writers have been observed to add extra newlines at this point, which is unnecessary with a stylesheet).

☞ *idem*, section 3.4.3.8, p. 187

TEST 8. Move a block of text from one place to another MOVE

We test the different ways in which mark–cut–scroll–find–move–paste can be replaced.

☞ section 3.4.3.6, p. 185; section 3.4.3.7, p. 186; section 4.3.5, p. 246

TEST 9. Highlighting a trade name (product name) INLINE

This tests inline visual markup (italic, bold, etc): the B, I, U buttons have a drop-down menu where the writer selects the reason why this font style is being selected.

☞ User Survey (section 3.4.3.5, p. 185); Modelling & Testing (section 4.3.4, p. 238; section 4.3.4.1, p. 240; Figure 4.11, p. 242)

TEST 10. Add a cross-reference to another section XREF

Similar to the Move Blocks test above, except the invocation of a cross-reference allows the writer to select the target visually by visiting it via a dialog or scroll action, and then return automatically to the point of reference.

☞ User Survey (section 3.4.3.9, p. 188); Modelling & Testing (section 4.3.6, p. 254)

TEST 11. Give a reference to an article you read CITE

Same method as for the preceding, except using a bibliographic database selection dialog to identify the target.

☞ *ibid*; section 4.3.6.3, p. 261

TEST 12. Insert a piece of another document INSDOC

In common with block moves and cross-referencing, this tests a dialog for identifying sections within external documents, so that the document does not have to be opened for copy-and-paste.

☞ Modelling & Testing (section 4.3.3.2, p. 236; section 4.3.7, p. 263)

Bibliography

Abolhassani, M., Fuhr, N., & Gövert, N. (2003). Information Extraction and Automatic Markup for XML Documents. In H. Blanken, T. Grabs, H. Schek, R. Schenkel, & G. Weikum (Eds.), *Intelligent search on XML data: applications, languages, models, implementations, and benchmarks.* Berlin: Springer.

Al-Awar, J., Chapanis, A., & Ford, W. (1981). Tutorials for the first-time computer user. *IEEE Transactions on Professional Communication*, *24*(1), 30-37.

Albanesius, C. (2010). In Blow to Microsoft, Validity of i4i Patent Upheld. *PC Magazine*. Retrieved from `http://www.pcmag.com/article2/0,2817,2367065,00.asp`

André, J., Brüggemann-Klein, A., Furuta, R., & Quint, V. (1994). *History of Document Processing.* `ftp://ftp.informatik.uni-freiburg.de/papers/brueggem/history.ps`.

André, J., Furuta, R., & Quint, V. (1989). By Way of an Introduction: Structured Documents: What and why? In J. André, R. Furuta, & V. Quint (Eds.), *Structured Documents* (p. 1-6). Cambridge: CUP.

Anghelache, R. (2004). The Meaning of Scientific Documents. In *New Developments in Electronic Publishing* (p. 5-7). Houston, TX: AMS/SMM.

Apple Corporation. (2008). *Apple Human Interface Guidelines* (Tech. Rep.). Cupertino, CA: Apple, Inc.

Apple Corporation. (2012). *OS X Human Interface Guidelines*. Cupertino, CA: Apple Corporation.

Arguello, J., Butler, B. S., Joyce, E., Kraut, R., Ling, K. S., Rosé, C., & Wang, X. (2006). Talk to me: Foundations for successful individual-group interactions in online communities. In *ACM SIGCHI Conference on Human Factors in Computing Systems, Montréal (22 April 2006)* (p. 959-968). New York, NY: ACM.

Bandor, M. S. (2006, Sep). *Quantitative Methods for Software Selection and*

Evaluation (Tech. Rep. No. CMU/SEI-2006-TN-026). Pittsburgh, PA: Carnegie-Mellon University, Software Engineering Institute.

Barfield, L. (1993). *The User Interface: Concepts and Design*. Reading, MA: Addison-Wesley.

Barnum, C. M. (2010). *Usability Testing Essentials: Ready, Set...Test*. Burlington, MA: Elsevier.

Baskette, F. K., Sissors, J. Z., & Brooks, B. S. (1986). *The Art of Editing*. New York: Macmillan.

Beeton, B. (1985). First principles of typographic design for document production (TUGboat, Vol. 5, No. 2, pp. 79–90): Corrigenda. *TUGboat*, *6*(1), 6.

Benson, C., Elman, A., Nickell, S., & Robertson, C. Z. (2012). *GNOME Human Interface Guidelines* (2.2.3 ed.). Boston, MA: Free Software Foundation. Retrieved from `https://developer.gnome.org/hig-book/hig-book-html-3.8.0.tar.gz`

Berg, N. E. (1975). *Electronic Composition: A Guide to the Revolution in Typesetting*. Alexandria, VA: Graphic Arts Technical Foundation.

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The Semantic Web. *Scientific American*(5), 34-43. Retrieved from `http://www.scientificamerican.com/article.cfm?id=the-semantic-web` doi: 10.1038/scientificamerican0501-34

Bevan, N. (2001). International Standards for HCI and Usability. *International Journal of Human Computer Studies*, *55*(4), 1-18.

Bevan, N. (2006). Practical issues in usability measurement. *Interactions*, *13*(6), 42-43. Retrieved from `http://0-doi.acm.org.library.ucc.ie/10.1145/1167948.1167976` doi: 10.1145/1167948.1167976

Bingham, H. (2001, May). *OASIS Technical Resolution Exchange Table Model Document Type Definition* (Tech. Rep.). Burlington, MA: Organization for the Advancement of Structured Information Standards.

Birnbaum, D. J. (1997, Jun). In Defense of Invalid SGML. In *Joint Annual Conference of the Association for Computers and the Humanities and the Association for Literary and Linguistic Computing, Queen's University, Kingston, ON (3 June 1997)*. Retrieved from `http://xml.coverpages.org/birnbaumACH97.html`

Birnbaum, D. J., Cournane, M., & Flynn, P. (1999, Apr). Using the TEI Writing System Declaration. *Computers and the Humanities*, *33*(1/2), 49-57. Retrieved from `doi:10.1023/A:1001742011783`

Birnbaum, D. J., & Mundie, D. A. (1999). The Problem of Anomalous Data: A Transformational Approach. *Markup Languages: Theory and Practice*, *1*(4), 1-19.

Bly, R. W., & Blake, G. (1982). *Technical writing: structure, standards, and style*. New York: McGraw-Hill.

Boice, R., & Jones, F. (1984). Why Academicians Don't Write. *The Journal of Higher Education*, *55*(5), 567-582. Retrieved from `http://www.jstor.org/stable/1981822`

Borenstein, N. S. (1985). The Evaluation of Text Editors: A Critical Review of the Roberts and Moran Methodology Based on New Experiments. In *CHI'85* (p. 99-105).

Borkin, S. A., & Prager, J. M. (1981). *Some issues in the design of an editor-formatter for structured documents* (Tech. Rep.). Cambridge, MA: IBM Cambridge Scientific Center, Cambridge, MA.

Born, G. (1995). *The File Formats Handbook*. Boston, MA: International Thompson Computing Press.

Bos, B., Lie, H. W., Lilley, C., & Jacobs, I. (2008, Apr). *Cascading Style Sheets, level 2* (Tech. Rep. No. REC-CSS2-20080411). Cambridge, MA: World Wide Web Consortium. Retrieved from `http://www.w3.org/TR/CSS2`

Bovair, S., Kieras, D. E., & Polson, P. G. (1990). The acquisition and performance of text-editing skill: a cognitive complexity analysis. *Human-Computer Interaction*, *5*, 1-48. Retrieved from `http://0-portal.acm.org.innopac.ucc.ie/citation.cfm?id=1455749.1455750&coll=Portal&dl=GUIDE&CFID=43258909&CFTOKEN=65816100`

Bradner, S. (1997, Mar). *Key words for use in RFCs to Indicate Requirement Levels* (Tech. Rep.). Reston, VA: Internet Engineering Task Force.

Bray, T., Paoli, J., Sperberg-McQueen, M., & Maler, E. (2000, Oct). *Extensible Markup Language Version 1.0* (2nd ed.; Tech. Rep.). Cambridge, MA: World Wide Web Consortium.

Bray, T., Sperberg-McQueen, M., Harvey, B., Beeton, B., & Price, L. (2009, Dec). *Earliest known structured-document editor*. Pers. Comm.

Buhler, R., & Buhler, S. (1990, May). P-Stat User's Manual (Vol. 1-3) [Computer software manual]. Princeton, NJ: Doktor Byte Press.

Burke, T. (1998). *Dewey's New Logic: A Reply to Russell*. Chicago, IL: University of Chicago Press.

Burnard, L. (1999). Is Humanities Computing an Academic Discipline? or, Why Humanities Computing Matters. In B. Nowviskie & J. Unsworth (Eds.), *"Is*

Humanities Computing an Academic Discipline?" —An Interdisciplinary Seminar. Charlottesville, VA: Institute for Advanced Technology in the Humanities. Retrieved from http://www.iath.virginia.edu/hcs/burnard.html

Burnard, L., & Sperberg-McQueen, M. (Eds.). (2007, Nov). *Guidelines for the Text Encoding Initiative* (P5 ed.; Tech. Rep.). Oxford: TEI Consortium.

Busa, R. (1980). The Annals of Humanities Computing: the *Index Thomisticus*. *Computers and the Humanities*, *14*, 83-90. doi: 10.1007/BF02403798

Card, S. K. (1978). *Studies in the psychology of computer text editing systems* (Doctoral dissertation, Carnegie-Mellon University and Xerox PARC). Retrieved from http://bitsavers.trailing-edge.com/pdf/xerox/parc/techReports/SSL-78-1_Studies_In_The_Psychology_Of_Computer_Text_Editing_Systems.pdf

Card, S. K., Moran, T. P., & Newell, A. (1980). The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, *23*(7), 396-410. doi: 10.1145/358886.358895

Card, S. K., Moran, T. P., & Newell, A. (1986). *The psychology of human-computer interaction*. Boca Raton, FL: Lawrence Erlbaum Associates.

Carroll, J. M. (Ed.). (1987). *Interfacing Thought: Cognitive aspects of Human-Computer Interaction*. Cambridge, MA: MIT Press.

Carroll, J. M. (2009). Human Computer Interaction. In M. Soegaard & R. Friis Dam (Eds.), *Encyclopedia of Human-Computer Interaction* (chap. 2). Aarhus, Denmark: The Interaction-Design.org Foundation. Retrieved from http://www.interaction-design.org/encyclopedia/human_computer_interaction_hci.html

Carroll, J. M., & Rosson, M. B. (2001). *Usability Engineering: Scenario-Based Development of Human-Computer Interaction* (1st ed.). San Diego, CA: Morgan Kaufmann/Academic Press.

Centre for the Study of Ancient Documents. (2003). Tab. Vindol. II 291. In *Vindolanda Tablets Online.* Oxford: Centre for the Study of Ancient Documents. Retrieved from http://vindolanda.csad.ox.ac.uk/TVII-291

Clark, J. (1992). *sgml-mode: SGML- and HTML-editing modes for GNU Emacs* (2007th ed.). Free Software Foundation, Inc. Boston, MA.

Clark, J. (2003). *nxml-mode: A new XML mode for GNU Emacs* (2004th ed.). Free Software Foundation, Inc. Boston, MA.

Coombs, J. H., Renear, A. H., & DeRose, S. J. (1987, Nov). Markup Systems and

The Future of Scholarly Text Processing. *Communications of the ACM*, *30*(11), 933-947.

Cooper, A. (1999). *The inmates are running the asylum: Why high-tech products drive us crazy and how to restore the sanity* (1st ed.). Indianapolis, IN: Sams.

Cooper, A. (2004). *The inmates are running the asylum: Why high-tech products drive us crazy and how to restore the sanity* (2nd ed.). Indianapolis, IN: Sams.

Cooper, A., & Reimann, R. (2003). *About Face 2.0*. Indianapolis, IN: Wiley.

Cooper, A., Reimann, R., & David, C. (2007). *About Face 3: The essentials of interface design* (3rd ed.). New York, NY: Wiley.

Cottrell, A. (1999). *Word Processors: Stupid and Inefficient*. `http://ricardo.ecn.wfu.edu/~cottrell/wp.html`.

Cournane, M. (1998). *The application of SGML/TEI to the processing of complex multilingual historical texts* (Unpublished doctoral dissertation). University College Cork, Department of History, Cork, Ireland.

Dau, F., & Sifer, M. (2007). A Formalism for Navigating and Editing XML Document Structure. *Databases in Networked Information Systems*, *4777*(11), 96-114.

Delsaerdt, P. (2011). Typographic design and renaissance lexicography: Cornelis Kiliaan's dictionaries of the Dutch language. *Journal of the Printing Historical Society*, *17*, 23-47. Retrieved from `https://lirias.kuleuven.be/handle/123456789/337282`

DeRose, S. J. (1997). *The SGML FAQ Book*. Boston, MA: Kluwer Academic.

DeRose, S. J. (2007, Aug). Markup Overlap: A Review and a Horse. In *Extreme Markup Languages, Montréal (7 August 2007)*. Oxford, UK: Literary and Linguistic Computing.

DeRose, S. J., Durand, D., Mylonas, E., & Renear, A. (1990, Winter). What is Text, Really? *Journal of Computing in Higher Education*, *2*(1), 3-26.

DeRose, S. J., & Vogel, J. (1995, Apr). *Data processing system and method for representing, generating a representation of and random access rendering of electronic documents* (No. US5557722). Electronic Book Technologies, Inc. Providence, RI.

Diaper, D., & Sanger, C. (2006). Tasks for and tasks in human–computer interaction. *Interacting with Computers*, *18*(1), 117-138. Retrieved from `http://www.sciencedirect.com/science/article/pii/S0953543805000603` doi: 10.1016/j.intcom.2005.06.004

Dix, A. (2010). Human–computer interaction: A stable discipline, a nascent

science, and the growth of the long tail. *Interacting with Computers*, *22*(1), 13-27. Retrieved from http://www.sciencedirect.com/science/article/pii/S0953543809000952 doi: 10.1016/j.intcom.2009.11.007

Doyle, A. C. (1892). *The Adventures of Sherlock Holmes: A Scandal in Bohemia*. London: George Newnes.

Drawehn, J., Altenhofen, C., Stanišić-Petrović, M., & Weisbecker, A. (2004, Apr). A Tool for Semi-automatic Document Reengineering. In *Reading and Learning* (p. 216-234). Springer. Retrieved from http://www.springerlink.com/content/0v5e3r3vrux7ml82/

Dumas, J. (2007). The great leap forward: the birth of the usability profession 1988-1993. *Journal of Usability Studies*, *2*(2), 54-60. Retrieved from http://www.designingforhumans.com/idsa/2007/02/a_brief_history.html

Eddington, A. S. (1920). *Space Time and Gravitation*. Cambridge: CUP.

Eliot, T. (1942). *Little Gidding*. London: Faber & Faber.

Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., & Ylonen, T. (1999, Sep). *SPKI Certificate Theory* (Tech. Rep.). Reston, VA: The Internet Society.

Enenkel, K. A., & Neuber, W. (2004). *Cognition and the Book: Typologies of Formal Organisation of Knowledge in the Printed Book of the Early Modern Period* (K. A. Enenkel & W. Neuber, Eds.). Leiden: Brill Academic. Retrieved from http://www.brill.com/cognition-and-book-typologies-formal-organisation-knowledge-printed-book-early-modern-period

Ensign, C. (1995, Dec). If SGML Is So Smart, How Come It Ain't Rich? In *SGML 95, Boston, MA (5 December 1995)*. Alexandria, VA: Graphic Communications Association.

Fahlgren, K. (2007). *XML Conf 2007 First Day*. http://www.oreillynet.com/xml/blog/2007/12/xml_conf_2007_first_day.html.

Fallside, D. C., & Walmsley, P. (2004, Oct). *XML Schema Part 0* (Tech. Rep.). Cambridge, MA: World Wide Web Consortium.

Faulkner, W. (1956, Mar). Jean Stein Interview: The Art of Fiction No. 12. *The Paris Review*(12), 5-7. Retrieved from http://www.theparisreview.org/interviews/4954/the-art-of-fiction-no-12-william-faulkner

Feller, J., & Fitzgerald, B. (2001). *Understanding Open Source Software Development*. Reading, MA: Addison-Wesley Professional.

Fenton, E. G., & Duggan, H. N. (2006, Sep). Effective Methods of Producing Machine-Readable Text from Manuscript and Print Sources. In L. Burnard, K. O'Brien O'Keeffe, & J. Unsworth (Eds.), *Electronic Textual Editing*

(p. 419). New York, NY: Modern Language Association of America.

Fine, J. (2001, Dec). Instant Preview and the TEX Dæmon. *TUGboat*, *22*(4), 292-298.

Flower, L., & Hayes, J. R. (1981). A Cognitive Process Theory of Writing. *College Composition and Communication*, *32*(4), 365-387. Retrieved from http://www.jstor.org/stable/356600

Flynn, P. (1993, Jul). TEX and SGML: A Recipe for Disaster? In *TUG 1993, Aston University, Birmingham (26 July 1993)* (p. 227-230). TEX Users Group. Retrieved November 10, 1999, from http://www.tug.org/TUGboat/Articles/tb14-3/tb40flynn.pdf

Flynn, P. (1997). W[h]ither the web? The extension or replacement of HTML. *Journal of the American Society for Information Science*, *48*(7), 614-621. Retrieved from http://doi.wiley.com/10.1002/(SICI)1097-4571(199707)48:7<614::AID-ASI8>3.0.CO;2-W doi: 10.1002/(SICI)1097-4571(199707)48:7<614::AID-ASI8>3.0.CO;2-W

Flynn, P. (1998). *Understanding SGML and XML Tools*. Boston: Kluwer.

Flynn, P. (2002). Formatting Information. *TUGboat*, *23*(2), 115-250. Retrieved from http://www.ctan.org/tex-archive/info/beginlatex/

Flynn, P. (2006, Aug). If XML is so easy, how come it's so hard?: The usability of editing software for structured documents. In *Extreme Markup 2006, Montréal (7 August 2006)*. IdeAlliance.org. Retrieved December 22, 2006, from http://epu.ucc.ie/articles/extreme06

Flynn, P. (2009, Aug). Why writers don't use XML: The usability of editing software for structured documents. In *Balisage 2009, Montréal (14 August 2009)* (Vol. 3). Rockville, MD: Balisage Series on Markup Technologies. doi: 10.4242/BalisageVol3.Flynn01

Flynn, P. (2013, Aug). Could authors really write in XML one day? In *Balisage 2013, Montréal (6 August 2013)* (Vol. 10). Rockville, MD: Balisage Series on Markup Technologies. doi: 10.4242/BalisageVol10.Flynn02

Furuta, R. (1992). Important papers in the history of document preparation systems: basic sources. *Electronic Publishing*, *5*(1), 19-44. Retrieved from http://cajun.cs.nott.ac.uk/compsci/epo/papers/volume5/issue1/ep057rf.pdf

Fyffe, C. (1969). *Basic Copyfitting*. London: Studio Vista.

Gaur, A. (1992). *A history of writing* (2nd ed.). London: British Library.

Gaver, W. W. (1991). Technology affordances. In (p. 79-84). New York, NY: ACM. doi: 10.1145/108844.108856

Geers, F. (2010). *User-friendly structured document editing: removing barriers for author acceptance* (Unpublished master's thesis). Utrecht University.

Gibson, J. J. (1977). The Theory of Affordances: Perceiving, Acting, and Knowing. In R. Shaw & J. Bransford (Eds.), . Hillsdale, NJ: Lawrence Erlbaum Associates.

Gibson, J. J. (1979). *The Ecological Approach to Visual Perception*. Boston, MA: Houghton Mifflin.

Gibson, N. (2013). *Just enough structure?* `http://www.corbas.co.uk/blog/2013/5/21/just-enough-structure`.

Goldfarb, C. (1990). *The SGML Handbook*. Oxford, England: OUP.

Goldfarb, C. (1996). *The roots of SGML —A personal recollection.* `http://www.sgmlsource.com/history/roots.htm`.

Goldfarb, C. (2003, Oct). A Brief History of the Development of SGML. In C. Goldfarb (Ed.), *The SGML History Niche.* Chesterbrook, PA: SGMLsource.com.

Goldfarb, C., Pepper, S., & Ensign, C. (1998). *SGML Buyer's Guide*. Upper Saddle River, NJ: Prentice Hall PTR.

Gould, J. D. (1987). *How to design usable systems* (Tech. Rep.). Yorktown Heights, NY: IBM Hawthorne.

Gould, J. D., & Lewis, C. (1983). Designing for usability –key principles and what designers think. In *CHI'83* (p. 50-53). New York, NY, USA: ACM. Retrieved from `http://0-doi.acm.org.library.ucc.ie/10.1145/800045.801579` doi: 10.1145/800045.801579

Gould, J. D., & Lewis, C. (1985). Designing for usability: key principles and what designers think. *Communications of the ACM, 28*(3), 300-311. Retrieved from `http://0-doi.acm.org.library.ucc.ie/10.1145/3166.3170` doi: 10.1145/3166.3170

Grafton, A. (1998). *The footnote: a curious history*. London: Faber.

Green, T. R., & Payne, S. J. (1984, Jun). Organization and learnability in computer languages. *International Journal of Man-Machine Studies, 21*(1), 7-18.

Green, T. R., Payne, S. J., & van der Veer, G. C. (Eds.). (1983). *The psychology of computer use*. London, England: Academic Press.

Haake, A., Huser, C., & Reichenberger, K. (1994, Jun). The Individualized Electronic Newspaper: an example of an active publication. *Electronic Publishing, 7*(2), 89-111.

Harper, R., Rodden, T., Rogers, Y., & Sellen, A. (2008). *Being Human:*

Human-Computer Interaction in the year 2020. Cambridge, England: Microsoft Research.

Hershey, A. V. (1972). A computer system for scientific typography. *Computer Graphics and Image Processing*, *1*(4), 373-385. Retrieved from `http://www.sciencedirect.com/science/article/pii/0146664X72900226` doi: 10.1016/0146-664X(72)90022-6

Hewett, T. T., & Meadow, C. T. (1986). On designing for usability: an application of four key principles. In *CHI'86* (p. 247-252). New York, NY, USA: ACM. Retrieved from `http://0-doi.acm.org.library.ucc.ie/10.1145/22627.22379` doi: 10.1145/22627.22379

Hilligoss, S., & Howard, T. (2002). *Visual Communication: a writer's guide* (2nd ed.). New York: Pearson Education (Longman).

Hoenicka, M. (2010, Nov). RefDB [Computer software manual].

Holman, K. (2013, Oct). Re: [xml-dev] Excellent quote from Len Bullard. *OASIS XML Developers' Mailing List Archives*, *201310*(57). Retrieved from `http://lists.xml.org/archives/xml-dev/201310/msg00057.html`

Hu, Z., Mu, S., & Takeichi, M. (2004). A Programmable Editor for Developing Structured Documents based on Bidirectional Transformations. In *PEPM'04* (p. 178-189). ACM. doi: 10.1145/1014007.1014025

Hurst, N., Li, W., & Marriott, K. (2009). Review of automatic document formatting. In *Proceedings of the 9th ACM Symposium on Document Engineering* (p. 99-108). Munich: ACM. Retrieved from `http://portal.acm.org/citation.cfm?doid=1600193.1600217` doi: 10.1145/1600193.1600217

Hutchins, E. L., Hollan, J. D., & Norman, D. A. (1985). Direct manipulation interfaces. *Human-Computer Interaction*, *1*(4), 311-338. Retrieved from `http://portal.acm.org/citation.cfm?id=1453235`

ISO JTC 1/SC 34. (1985). *Standard Generalized Markup Language* (Tech. Rep.). Geneva: International Organization for Standardization.

ISO TC 159/SC 4. (1999). *Human-centred design processes for interactive systems* (Tech. Rep.). Geneva: International Organization for Standardization.

ISO TC 159/SC 4. (2010). *Ergonomics of human-system interaction —Part 210: Human-centred design for interactive systems* (Tech. Rep.). Geneva: International Organization for Standardization.

Jelliffe, R. (2008). The design goals of XML. *O'Reilly XML Blog*, *2008*(01-06). Retrieved from `http://www.oreillynet.com/xml/blog/2008/01/`

the_design_goals_of_xml_1.html

John, B. E., & Kieras, D. E. (1996, Dec). The GOMS family of user interface analysis techniques: comparison and contrast. *ACM Transactions on Computer-Human Interaction, 3*(4), 320-351.

Johnson, J. (2000). *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers* (S. Card, J. Grudin, J. Nielsen, & T. Skelly, Eds.). San Francisco: Morgan Kaufmann.

Jørgensen, A. H., Barnard, P., Hammond, N., & Clark, I. (1983). Naming commands: an analysis of designers' naming behaviour. In T. R. G. Green, S. J. Payne, & G. C. van der Veer (Eds.), *The psychology of computer use* (p. 69-88). London: Academic Press.

Kastrup, D. (2002). Revisiting WYSIWYG Paradigms for Authoring in LaTeX. *TUGboat*, *23*(1), 57-64. Retrieved from http://tug.org/TUGboat/tb23-1/kastrup.pdf

Kay, M. (2013, Oct). Re: [the XML Guild] Implied semantics of element type names. *XML Guild mailing list*. Retrieved from Author's archive

Keith, S. (1995, Dec). Creating DTDs via the GB-Engine and Fred. In *SGML '95, Boston, MA (5 December 1995)* (p. 399-404). Alexandria, VA: Graphic Communications Association.

Kelly, D., & Abrahamson, D. (1991, Sep). Document Structure Recognition. In *First International Conference on Document Analysis and Recognition, St. Malo, France (30 September 1991)* (p. 525-532). Sophia Antipolis: Association française pour la cybernétique économique et technique, now Société informatique de France.

Kernighan, B. W. (1978). *A TROFF Tutorial* (Tech. Rep.). Murray Hill, NJ: Bell Labs. Retrieved from http://www.kohala.com/start/troff/v7man/trofftut/trofftut.ps

Kernighan, B. W., & Cherry, L. L. (1975). A system for typesetting mathematics. *Communications of the ACM*, *18*(3), 151-157. Retrieved from http://doi.acm.org/10.1145/360680.360684 doi: 10.1145/360680.360684

Khalili, A., & Auer, S. (2013a). User Interfaces for Semantic Authoring of Textual Content: A Systematic Literature Review. *Journal of Web Semantics*, *20*, 26.

Khalili, A., & Auer, S. (2013b). WYSIWYM —Integrated Visualization , Exploration and Authoring of Unstructured and Semantic Content. *Semantic Web Journal*, *4*(3), 11. Retrieved from http://www.semantic-web-journal.net/content/

wysiwym-integrated-visualization-exploration-and-authoring-un-structured-and-semantic

Khalili, A., Auer, S., & Hladky, D. (2012). The RDFa Content Editor —From WYSIWYG to WYSIWYM. In *2012 IEEE 36th Annual Computer Software and Applications Conference* (p. 531-540). İzmir, Turkey: IEEE. Retrieved from http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6340208 doi: 10.1109/COMPSAC.2012.72

Kirakowski, J. (1988). *Human Computer Interaction: from Voltage to Knowledge*. Bromley, UK: Chartwell-Bratt Ltd. Retrieved from http://www.amazon.com/dp/0862381797

Kirakowski, J. (2008). State-of-the-Art Introduction to HCI Usability. In *iHCI Conference.*

Kirakowski, J., & Bevan, N. (1997). MAPI —MUSiC Assisted Process Improvement. In K. Varghese & S. Pfleger (Eds.), *Human comfort and security of information systems.* Berlin: Springer.

Kirakowski, J., & Corbett, M. (1990). *Effective Methodology for the Study of HCI:* (No. 5). Amsterdam: North-Holland.

Kirakowski, J., & Corbett, M. (1993). SUMI: the Software Usability Measurement Inventory. *British Journal of Educational Technology*, *24*(3), 210-212. doi: 10.1111/j.1467-8535.1993.tb00076.x

Kirakowski, J., & Murphy, R. (2009). A comparison of current approaches to usability measurement. In *I-HCI 2009* (p. 13-17). Dublin: I-HCI 2009, University College Dublin.

Kirschenbaum, V. (2005). *Goodbye Gutenberg*. New York: Global Renaissance Society.

Kleinfeld, S. (2013, Aug). The Case for Authoring and Producing Books in (X)HTML5. In *Balisage, Montréal (6 August 2013)* (Vol. 10). Rockville, MD: Balisage Series on Markup Technologies. doi: 10.4242/BalisageVol10.Kleinfeld01

Knuth, D. E. (1986). *The TEXbook* (Vol. A). Reading, MA: Addison-Wesley.

Knuth, D. E. (1997). *Fundamental Algorithms* (3rd ed., Vol. I). Reading, MA: Addison-Wesley.

Knuth, D. E., Larrabee, T., & Roberts, P. M. (1989). *Mathematical writing*. Providence, RI: The Mathematical Association of America.

Krug, S. (2006). *Don't make me think: A common sense approach to web usability* (2nd ed.). Berkeley, CA: New Riders.

Laforest, F., & Flory, A. (2002). Using weakly structured documents at the

user-interface level to fill in a classical database. In K. Siau (Ed.), (Vol. 1, p. 190-210). Hershey, PA, USA: IGI Global. Retrieved from http://dl.acm.org/citation.cfm?id=960129.960140

Lamport, L. (1986). *LaTeX: A Document Preparation System* (1st ed.). Reading, MA: Addison-Wesley.

Lamport, L. (1994). *LaTeX: A Document Preparation System* (2nd ed.). Reading, MA: Addison-Wesley.

Lamport, L. (2000). How LaTeX changed the face of mathematics. *DMV-Mitteilungen*(1), 49-51.

Landauer, T. (1995). *The Trouble with Computers: Usefulness, Usability, and Productivity*. Cambridge, MA: MIT Press.

Lapeyre, D., & Piez, W. (2007). Introductory Schematron. In *XML Conference*. Boston, MA: IdeAlliance.

Larivière, V., Gingras, Y., & Archambault, É. (2006). Canadian collaboration networks: A comparative analysis of the natural sciences, social sciences and the humanities. *Scientometrics*, *68*(3), 519-533. Retrieved from http://dx.doi.org/10.1007/s11192-006-0127-8 doi: 10.1007/s11192-006-0127-8

Larsson, J., & Kastrup, D. (2007). Preview-LaTeX [Computer software manual]. Boston, MA: Free Software Foundation.

Laurel, B., & Mountford, S. J. (1990). *The Art of human-computer interface design*. Addison-Wesley.

Laurens, J. (2007, Aug). Will TeX ever be WYSIWYG? or, the PDF synchronization story. *The Practical TeX Journal*, *3*, 8. Retrieved from http://tug.org/pracjourn/2007-3/laurens/

Lawson, A. (1990). *Anatomy of a typeface*. London: Hamish Hamilton (Penguin).

Legat, M. (1986). *Writing for pleasure and profit*. London: Robert Hale.

Lesk, M. E. (1976). *Tbl —a program to format tables* (Tech. Rep. No. 49). Murray Hill, NJ: Bell Laboratories.

Lesk, M. E. (1978). *Some applications of inverted indexes on the UNIX system* (Tech. Rep. No. 69). Murray Hill, NJ: Bell Laboratories.

Levi, J. (2013). *Why text software is outdated: my thoughts on why (and how) typing could be better.* https://medium.com/future-tech-future-market/265d23d91254.

Lewin, K. (1926). *Vorsatz, Wille und Bedürfnis. Mit Vorbemerkungen über die psychischen Kräfte und Energien und die Struktur der Seele*. Berlin: Springer.

Lewis, J. R. (2012). Usability Testing. In G. Salvendy (Ed.), *Handbook of Human*

Factors and Ergonomics (4th ed., p. 1267-1312). New York: Wiley.

Licklider, J. C. R. (1960). Man-Computer Symbiosis. *Human Factors in Electronics, Institute of Radio Engineers Transactions on Human Factors in Electronics*, *HFE-1*(1), 4 -11. doi: 10.1109/THFE2.1960.4503259

Maass, S. (1983). Why systems transparency? In T. R. G. Green, S. J. Payne, & G. C. van der Veer (Eds.), *The psychology of computer use* (p. 19-28). London: Academic Press.

Mackichan, B. (2007). Design Decisions for a Structured Front End to LaTeX Documents. In B. Beeton (Ed.), *TEX Users Group Conference* (p. 58-66). Portland, OR: TEX Users Group.

Madnick, S. E., & Moulton, A. (1968). SCRIPT: an on-line manuscript processing system. *IEEE Transactions on Engineering Writing and Speech*, *11*(2), 92-100.

Maler, E., & el Andaloussi, J. (1999). *Developing SGML DTDs: from Text to Model to Markup*. Upper Saddle River, NJ: Prentice-Hall.

Marcoux, Y., & Rizkallah, É. (2009). Intertextual Semantics: A Semantics for Information Design. *Journal of the American Society for Information Science and Technology*, *60*(9), 1895-1906. doi: 10.1002/asi.21134

Marr, D. (1982). *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. San Francisco, CA: WH Freeman.

Marsh, J., Daniel, V., & Norman, W. (2005, Sep). *xml:id Version 1.0* (2nd ed.; Tech. Rep.). Cambridge, MA: World Wide Web Consortium.

Mayhew, D. J. (2005). Keystroke level modelling as a cost-justification tool. In R. G. Bias & D. J. Mayhew (Eds.), *Cost-Justifying Usability: An Update for the Internet Age, Second Edition* (2nd ed., p. 465-488). San Francisco, CA: Morgan Kaufmann.

McCarthy, W. (1999). What is humanities computing? Toward a Definition of the Field. In B. Nowviskie & J. Unsworth (Eds.), *"Is Humanities Computing an Academic Discipline?" —An Interdisciplinary Seminar*. Charlottesville, VA: Institute for Advanced Technology in the Humanities. Retrieved from `http://ilex.cc.kcl.ac.uk/wlm/essays/what/`

McGregor, D. (1960). *The Human Side of Enterprise*. Princeton, NJ: McGraw-Hill.

McNamara, N., & Kirakowski, J. (2006). Functionality, Usability, and User Experience: Three Areas of Concern. *Interactions*, *Nov-Dec*, 26-28.

Microsoft Corporation. (1995). *The Windows Interface Guidelines for Software Design: An Application Design Guide* (2nd ed.). Redmond, WA. Microsoft

Press.

Microsoft Corporation. (2008). Microsoft Commends Chinese Court in Sentencing Ringleaders of World's Largest Software Counterfeiting Syndicate. *Microsoft News Center*, 4. Retrieved from http://www.microsoft.com/presspass/press/2008/dec08/12-31ChinaSentencingPR.mspx

Microsoft Corporation. (2010). *Windows User Experience Interaction Guidelines for Windows 7 and Windows Vista*. Redmond, WA: Microsoft Corporation. Retrieved from http://www.microsoft.com/en-us/download/confirmation.aspx?id=2695

Moroney, M. J. (1984). *Facts from Figures* (2nd ed.). London, UK: Penguin.

Murphy, D. (2009). The Beginnings of TECO. *IEEE Annals of the History of Computing*, *31*(4), 110-115. Retrieved from http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5370787 doi: 10.1109/MAHC.2009.127

Murray, T. (2003). MetaLinks: Authoring and Affordances for Conceptual and Narrative Flow in Adaptive Hyperbooks. *International Journal of Artificial Intelligence in Education*, *13*(2), 199-233. Retrieved from http://iospress.metapress.com/content/PG0UWF2L5HH6729A

Murray-Rust, P. (2008, May). The merits and demerits of PDF: Debate with Chris Rusbridge (Digital Curation Centre) on Science publishing, workflow, PDF and Text Mining. *A Scientist and the Web (blog at Unilever Centre for Molecular Informatics)*. Retrieved from http://blogs.ch.cam.ac.uk/pmr/2008/05/03/the-merits-and-demerits-of-pdf/

Negroponte, N. (2012). EmTech Prism: Education and Learning. In *Emerging Technologies.* Cambridge, MA: MIT. Retrieved from http://www2.technologyreview.com/emtech/12/agenda/

Neumann, R. K. (2001). *Legal reasoning and legal writing: structure, strategy, and style*. Gaithersburg, MD: Aspen Publishers.

Nielsen, J. (1994). Usability Inspection Methods. In *CHI'94* (p. 413-414). Boston, MA: ACM.

Nielsen, J. (2000). *Designing Web Usability* (1st ed.). New Riders Publishing. Retrieved from http://www.amazon.com/dp/156205810X

Nielsen, J., & Molich, R. (1990). Heuristic evaluation of user interfaces. In *CHI'90* (p. 249-256). ACM. doi: 10.1145/97243.97281

Nikšić, H., & Cowan, M. (2005, Oct). GNU Wget 1.10.2 [Computer software manual]. Boston, MA: Free Software Foundation.

Norman, D. A. (1986). *The Design of Everyday Things*. New York, NY: Basic Books.

Norman, D. A. (1988). *The Design of Everyday Things* (formerly *The Psychology of Everyday Things*). New York: Basic Books.

Norman, D. A. (1999). Affordance, conventions, and design. *Interactions*, *6*(3), 38-43. doi: 10.1145/301153.301168

Norman, D. A. (2004). *Emotional Design*. Cambridge, MA: Basic Books.

Norman, D. A., & Draper, S. W. (1986). *User centered system design: new perspectives on human-computer interaction*. Mahwah, NJ: Lawrence Erlbaum Associates.

Norman, D. A., Miller, J., & Henderson, A. (1995). What You See, Some of What's in the Future, And How We Go About Doing It: HI at Apple Computer. In *CHI'95* (p. 155). Denver, CO: ACM. doi: 10.1145/223355.223477

Norrish, P. (1989). Semantic Structures of Text. In J. André, R. Furuta, & V. Quint (Eds.), *Structured Documents* (p. 143-159). Cambridge, England: CUP.

O'Connor, C., Gnanapiragasam, A., & Hepp, M. (2013, Aug). ProofExpress: An online browser-based XML article proofing system for STM journals. In *Balisage International Symposium on Native XML user interfaces, Montréal (5 August 2013)*. Rockville, MD: Balisage Series on Markup Technologies. doi: 10.4242/BalisageVol11.OConnor01

O'Keeffe, T. J. (1987). *A comparison of human performance on computer text editing tasks using windowed and non-windowed strategies* (Doctoral dissertation). Retrieved from `http://search.proquest.com/docview/303616312`

Olson, J. R., & Olson, G. M. (1990). The growth of cognitive modeling in human-computer interaction since GOMS. *Human-Computer Interaction*, *5*(2), 221-265. doi: 10.1207/s15327051hci0502&3_4

Orlandi, T. (2002). Is Humanities computing a discipline? *Jahrbuch für Computerphilologie*, *4*, 51-58. Retrieved from `http://computerphilologie.uni-muenchen.de/jg02/orlandi.html`

Ossanna, J. F., & Kernighan, B. W. (1976). *Troff User's Manual* (Tech. Rep. No. 54). Murray Hill, NJ: Bell Laboratories.

Owston, R. D., Murphy, S., & Wideman, H. H. (1992). The Effects of Word Processing on Students' Writing Quality and Revision Strategies. *Research in the Teaching of English*, *26*(3), 249-276. Retrieved from `http://www.jstor.org/stable/40171308`

Papanek, V. (1972). *Design for the Real World: Human ecology and social change*. London: Thames and Hudson.

Paul, R. (2008). Norwegian standards body implodes over OOXML controversy. *Ars Technica*. Retrieved from http://arstechnica.com/uncategorized/2008/10/norwegian-standards-body-implodes-over-ooxml-controversy/

Payne, S. J. (1991, Sep). Display-based action at the user interface. *International Journal of Man-Machine Studies*, *35*(3), 275-289.

Payne, S. J., & Green, T. R. (1986). Task-Action Grammars: A Model of the Mental Representation of Task Languages. *Human-Computer Interaction*, *2*, 93-133.

Piez, W., & Usdin, T. (2007). Separating Mapping from Coding in Transformation Tasks. In *XML Conference.* Boston, MA: IdeAlliance.

Pilditch, J. (1976). *Talk about Design*. London: Barrie and Jenkins.

Pirsig, R. (1974). *Zen and the Art of Motorcycle Maintenance: An Inquiry Into Values*. London: Bodley Head.

Plotnik, A. (Ed.). (1984). *The Elements of Editing: A Modern Guide for Editors and Journalists*. New York: Collier Macmillan.

Quark. (2012). *How Quark XML Author Enhances Microsoft Word* (Tech. Rep.). Denver, CO: Quark, Inc.

Quellmalz, E. S. (1982). *Scale for evaluating expository writing* (Vol. 26; Tech. Rep. No. 3). Washington, DC: Center for the Study of Evaluation. ERIC Document Reproduction Service No. ED236670. Retrieved from http://www.jstor.org/stable/40171308

Quin, L. (2013, Jun). Re: [xml-dev] Re: Native XML Interfaces. *comp.text.tex* (1370046128.13873.33.camel@localhost), (all pages).

Quint, V. (1989). Systems for the Manipulation of Structured Documents. In J. André, R. Furuta, & V. Quint (Eds.), *Structured Documents* (p. 39-74). Cambridge: CUP.

Quint, V., & Vatton, I. (1986). GRiF: an interactive System for Structured Document Manipulation. In J. C. van Vliet (Ed.), *International Conference on Text Processing and Document Manipulation* (p. 200-213). Cambridge, UK: Cambridge University Press.

Quint, V., & Vatton, I. (2004). Techniques for Authoring Complex XML Documents. Milwaukee, WI: ACM. Retrieved from http://wam.inrialpes.fr/publications/2004/DocEng2004VQIV.html

Quint, V., & Vatton, I. (2007). Structured templates for authoring semantically rich documents. In *Proceedings of the 2007 international workshop on Semantically aware document processing and indexing - SADPI '07* (p. 41-48).

New York, NY: ACM Press. Retrieved from http://dl.acm.org/citation.cfm?doid=1283880.1283889 doi: 10.1145/1283880.1283889

Raggett, D., Le Hors, A., & Jacobs, I. (Eds.). (1999, Dec). HTML 4.01 Specification [Computer software manual]. Boston, MA: W3C. Retrieved from http://www.w3.org/TR/html401/

Raggett, D., Teague, T., Hoehrmann, B., Reitzel, C., & Vela, M. (2004, Dec). HTML Tidy 0.99.0 [Computer software manual]. Mountain View, CA: Sourceforge.

Raymond, E. (2004, Jan). Real Programmer. *The Jargon File*. Retrieved from http://www.catb.org/jargon/html/R/Real-Programmer.html

Reid, B. K. (1980a). A high-level approach to computer document formatting. In (p. 24-31). New York, NY, USA: ACM. doi: 10.1145/567446.567449

Reid, B. K. (1980b). *Scribe: A Document Specification Language and its Compiler* (Unpublished doctoral dissertation). Carnegie-Mellon University.

Reid, B. K. (1989). Electronic Mail of Structured Documents: Representation, transmission, and archiving. In J. André, R. Furuta, & V. Quint (Eds.), *Structured Documents* (p. 107-118). Cambridge: CUP.

Relles, N., & Price, L. A. (1981). A user interface for online assistance. In (p. 400-408). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=800078.802553

Renear, A., Mylonas, E., & Durand, D. (1996). Refining Our Notion of What Text Really Is: The Problem of Overlapping Hierarchies. In N. Ide & S. Hockey (Eds.), *Joint Annual Conference of the Association for Computers and the Humanities and the Association for Literary and Linguistic Computing, Oxford (5 April 1992)* (Vol. 4). Clarendon Press.

Roberts, T. L. (1979). *Evaluation of Computer Text Editors* (Doctoral dissertation, Stanford University and Xerox PARC). Retrieved from Edinburgh University Library

Roberts, T. L., & Moran, T. P. (1982). Evaluation of text editors. In *Proceedings of the 1982 conference on Human factors in computing systems - CHI '82* (p. 136-141). New York: ACM Press. Retrieved from http://portal.acm.org/citation.cfm?doid=800049.801770 doi: 10.1145/800049.801770

Rubin, J. (1994). *Handbook of Usability Testing*. New York: John Wiley.

Rusty Harold, E. (2007, Jun). Correct, beautiful, fast (in that order): Lessons from designing XML verifiers. In A. Oram & G. Wilson (Eds.), *Beautiful Code* (p. 59-74). Sebastopol, CA: O'Reilly.

Saltzer, J. H. (1964). *TYPSET and RUNOFF: Memorandum editor and type-out commands* (Tech. Rep. No. MIT Computation Center CC-244 (Project MAC MAC-M-193)). Cambridge, MA: MIT.

Sampson, C. R. (1996). SASOUT: a context-based table model. In *SGML'96 Conference Proceedings (Boston, MA)* (p. 235-263). Alexandria, VA: Graphic Communications Association. Retrieved from `http://xml.coverpages.org/sasoutTableDTD.txt`

Saunders, J., & O'Malley, C. D. (1950). *The Illustrations from the Works of Andreas Vesalius of Brussels*. Cleveland, OH: World Publishing Co. Retrieved from `http://www.questia.com/PM.qst?a=o&d=6540216`

Scapin, D. L. (1982). Computer commands labelled by users versus imposed commands and the effect of structuring rules on recall. In *CHI'82* (p. 17-19). Gaithersburg, MD: ACM. doi: 10.1145/800049.801747

Schmidt, D. (2010). The inadequacy of embedded markup for cultural heritage texts. *Literary and Linguistic Computing*.

Schneidermann, B. (2003). Supporting Creativity with Advanced Information-Abundant User Interfaces. In B. Bedersen & B. Schneidermann (Eds.), *The Craft of Information Visualization: Readings and Reflections* (p. 372-377). New York: Morgan Kaufman.

Schumacher, E. F. (1973). *Small is beautiful: economics as if people mattered*. New York: Harper and Row.

ScooterTex. (2010, Jun). Is Something Broken? USENET search broken. *Google Groups Help Forum*. Retrieved from `http://groups.google.com/group/is-something-broken/browse_thread/thread/e6c17b7e2268acf9/678080d4009ab839?show_docid=678080d4009ab839`

Shackel, B. (1959). Ergonomics for a computer. *Design*, *120*, 36-39.

Shackel, B., & Richardson, S. J. (1991). *Human Factors for Informatics Usability*. Cambridge University Press.

Sherman, C. R. (1995). *Imagining Aristotle: Verbal and Visual Representation in Fourteenth-Century France*. Berkeley, CA: University of California Press.

Sieckenius de Souza, C., Prates, R. O., & Carey, T. (2000). Missing and Declining Affordances: Are these Appropriate Concepts? *Journal of the Brazilian Computer Society*, *7*(1), 26-34.

Sifer, M., Peres, Y., & Maarek, Y. (2002). A grammar view for editing structured documents. In *VIP2001* (Vol. 11, p. 23-24). Sydney: Australian Computer Society. Retrieved from `http://0-dl.acm.org.library.ucc.ie/citation.cfm?id=858375`

.858380&coll=DL&dl=ACM&CFID=326989324&CFTOKEN=74273963

Snyder, C. (2003). *Paper Prototyping*. San Francisco: Morgan Kaufmann (Elsevier Science).

Southall, R. (1984). First principles of typographic design for document production. *TUGboat*, *5*(2), 79-90. (As corrected in Beeton (1985))

Southall, R. (1989). Interfaces between the Designer and the Document. In J. André, R. Furuta, & V. Quint (Eds.), *Structured Documents* (p. 119-131). Cambridge, England: CUP.

Sperberg-McQueen, C. M. (2007a, Aug). Rabbit/duck grammars: a validation method for overlapping structures. In *Extreme Markup Languages, Montréal (7 August 2007)*. Oxford, UK: IdeAlliance.

Sperberg-McQueen, C. M. (2007b, Aug). Representation of overlapping structures. In *Extreme Markup Languages, Montréal (7 August 2007)*.

Sperberg-McQueen, C. M., Huitfeld, C., & Renear, A. (2000). Meaning and interpretation of Markup. *Markup Languages*, *2*(3), 215-234.

Sperberg-McQueen, C. M., & Huitfeldt, C. (1998, Jul). Concurrent Document Hierarchies in MECS and SGML. In *Association for Computers and the Humanities and the Association for Literary and Linguistic Computing, Debrecen, Hungary (5 July 1998)*. Oxford, UK: Literary and Linguistic Computing. Retrieved July 22, 2006, from `http://xml.coverpages.org/sperbergACH98.html`

Sperberg-McQueen, C. M., & Huitfeldt, C. (1999, Jun). GODDAG: A Data Structure for Overlapping Hierarchies . In *Association for Computers and the Humanities and the Association for Literary and Linguistic Computing, Charlottesville, VA (12 June 1999)*. Oxford, UK: Literary and Linguistic Computing.

Spinellis, D. (2007, Jun). Another level of indirection. In A. Oram & G. Wilson (Eds.), *Beautiful Code* (p. 279-291). Sebastopol, CA: O'Reilly.

Staflin, L. (1992). *psgml-mode: SGML-editing mode with parsing support for GNU Emacs* (2005th ed.). Free Software Foundation, Inc. Boston, MA.

Stallman, R. M. (1981, Oct). EMACS: The Extensible, Customizable Display Editor. In *ACM Conference on Text Processing*.

Stallman, R. M. (1985). *The GNU Manifesto*. Cambridge, MA: Free Software Foundation.

Stephani, P. (2009, Sep). Re: `\begin{section}` `\end{section}`. *comp.text.tex* (4aa2ba2c$0$30220$9b4e6d93@newsspool1.arcor-online.net), (all pages).

Strange, J. (2003). Organizing, Editing, and Linking Content. In W. Kasdorf (Ed.), *The Columbia Guide to Digital Publishing* (p. 155-178). New York: Columbia University Press.

Taghva, K., Condit, A., & Borsack, J. (1995, Feb). An Evaluation of an Automatic Markup System. In *SPIE'95: Document Recognition II, San Jose, CA (5 February 1995)* (p. 317-327). Washington, DC: USDOE.

Tamir, D., Komogortsev, O. V., & Mueller, C. J. (2008). An Effort and Time Based Measure of Usability. In *Proceedings of the 6th International Workshop on Software Quality* (p. 47-52). Leipzig: ACM.

Tannahill, A. (2008). *Publishing for success: a practical guide*. Belfast, NI: NIPR.

Tazi, S. (1989). Performing Textual Structures in Documents. In J. André, R. Furuta, & V. Quint (Eds.), *Structured Documents* (p. 180-189). Cambridge, England: CUP.

Tesler, L. (1972). *PUB: the document compiler* (Tech. Rep.). Stanford, CA: Stanford Artificial Intelligence Project. Retrieved from `http://www.nomodes.com/pub_manual.html`

The GNOME Usability Project. (2008). *Gnome Human Interface Guidelines* (2.2nd ed.; Tech. Rep.). Cambridge, MA: Gnome Foundation.

Thompson, H. (Ed.). (2000, Jan). *XED: An XML document instance editor* (Tech. Rep.). Edinburgh: University of Edinburgh, HCRC Language Technology Group. Retrieved from `http://www.ltg.ed.ac.uk/~ht/xed.html`

Tognazzini, B. (1992). *TOG on Interface*. Reading, MA: Addison-Wesley.

Toleman, M., & Welsh, J. (1994). An evaluation of editing paradigms. In S. Howard & Y. K. Leung (Eds.), *1994 Australasian Computer-Human Interaction Conference: Harmony Through Working Together (OzCHI'94)* (p. 73-78). Downer, ACT, Australia: CHISIG Australia. Retrieved from `http://www.ergonomics.org.au/downloads/archived_ozchi_conf_proceedings/OZCHI_1994.pdf`

Tschichold, J. (1951). *Designing Books: Planning a book; a typographer's composition rules; fifty-eight examples by the author*. New York, NY: Wittenborn, Schultz.

Turabian, K. L. (1973). *A manual for writers of term papers, theses, and dissertations*. Chicago, IL: University of Chicago Press.

Tyler, R. W., Gagné, R. M., & Scriven, M. (1967). *Perscpectives of Curriculum Evaluation* (Tech. Rep.). Chicago, IL: Rand McNally.

Underhill, P. (1999). *Why We Buy: The Science of Shopping–Updated and Revised*

for the Internet, the Global Consumer, and Beyond. New York: Simon & Schuster.

van den Broek, T. (2005, Jan). Choosing an XML editor. *Arts and Humanities Data Service*.

Walker, S. (1983). *Descriptive techniques for studying verbal graphic language* (Unpublished doctoral dissertation). University of Reading, Department of Typographic and Graphic Communication. (Cited in Southall (1984))

Waller, R. (1980). Graphic aspects of complex texts: typography as macro-punctuation. In P. Kolers & M. W. and (Eds.), *Processing of visible language.* New York: Plenum.

Walsh, N. (2001, Aug). *XML Catalogs* (Tech. Rep. No. spec-2001-08-06). Billerica, MA: Organization for the Advancement of Structured Information Standards. Retrieved from `http://www.oasis-open.org/committees/entity/spec-2001-08-06.html`

Walsh, N., & Muellner, L. (1999). *DocBook: The Definitive Guide*. Sebastopol, CA: O'Reilly. Retrieved from `http://www.docbook.org/tdg/en/html/ch05.html#ch05-layers`

Weiser, E. B. (2001, Dec). The Functions of Internet Use and Their Social and Psychological Consequences. *CyberPsychology and Behavior*, *4*(6), 723-743.

Werby, O. (2010). *The History of Usability*. `http://www.interfaces.com/blog/2010/09/the-history-of-usability/`.

Wharton, C., Rieman, J., Lewis, C., & Polson, P. (1994). The cognitive walkthrough method: a practitioner's guide. In J. Nielsen & R. Mack (Eds.), *Usability inspection methods* (p. 45). New York: Wiley & Sons. Retrieved from `http://www.colorado.edu/ics/node/521/attachment`

Wildman, D. (1995). Getting the most from paired-user testing. *Interactions*(July), 21-27. doi: 10.1145/208666.208675

Williams, R. (1990). *The Mac is not a typewriter: a style manual for creating professional-level type on your Macintosh*. Berkeley, CA: Peachpit Press.

Williams, R. (1992). *The PC is not a typewriter: a style manual for creating professional-level type on your personal computer*. Berkeley, CA: Peachpit Press.

Wilson, P. (2007, Oct). Between Then and Now. *TUGboat*, *28*(3), 280-298. Retrieved from `http://tug.org/TUGboat/Articles/tb28-3/tb90wilson.pdf`