Exploration of the relationship between tacit knowledge and software system test complexity

Niall Geary

109223354

Thesis Submitted to the National University of Ireland, Cork for the Degree of Doctor of Philosophy

Department of Accounting, Finance and Information Systems

Head of Department: Prof. Ciaran Murphy

Supervisors: Prof. Frédéric Adam
Dr. Tom O'Kane

September 2016

TABLE OF CONTENTS

List of Tables

List of Figures

The author declares that, except where duly acknowledged, that this thesis is my own work, and I have not obtained a degree in this university or elsewhere on the basis of the work submitted in this thesis.

Abstract

This research has explored the relationship between system test complexity and tacit knowledge. It is proposed as part of this thesis, that the process of system testing (comprising of *test planning*, *test development*, *test execution*, *test fault analysis*, *test measurement*, and *test case management*), is directly affected by both complexity associated with the system under test, and also by other sources of complexity, independent of the system under test, but related to the wider process of system testing. While a certain amount of knowledge related to the system under test is inherent, tacit in nature, and therefore difficult to make explicit, it has been found that a significant amount of knowledge relating to these other sources of complexity, can indeed be made explicit.

While the importance of explicit knowledge has been reinforced by this research, there has been a lack of evidence to suggest that the availability of tacit knowledge to a test team is of any less importance to the process of system testing, when operating in a traditional software development environment. The sentiment was commonly expressed by participants, that even though a considerable amount of explicit knowledge relating to the system is freely available, that a good deal of knowledge relating to the system under test, which is demanded for effective system testing, is actually tacit in nature (approximately 60% of participants operating in a traditional development environment, and 60% of participants operating in an agile development environment, expressed similar sentiments). To cater for the availability of tacit knowledge relating to the system under test, and indeed, both explicit and tacit knowledge required by system testing in general, an appropriate knowledge management structure needs to be in place. This would appear to be required, irrespective of the employed development methodology.

Acknowledgements

I am extremely grateful to Professor Frédéric Adam and Dr. Tom O'Kane for facilitating this research and sharing their considerable wealth of knowledge. I learned more than I could ever have imagined.

I would like to thank EMC Corporation (EMC²), SQS Software Quality Systems AG, Delaware Life, and CoreHR, for their enthusiastic participation in this research, and affording me invaluable access to willing participants.

Finally, it would be remiss of me not to mention my family and the sacrifices they have made to allow me devote time to this research.

1 Introduction

In 2009, the Standish Group released their CHAOS report stating that software development project success rates were running at 32%, outright failures were listed as 24% and 44% of projects were categorised as "challenged" projects. 68% of projects were either cancelled or seriously over-budget, behind schedule, or short some requirements (Standish Inc., 2009). A number of other authors have acknowledged the importance of the identified system development project goals of adhering to project schedule objectives, adhering to cost objectives, and meeting predefined requirements objectives ((Berman & Cutler, 1998), (Liu, Chen, Chan, & Lie, 2008), (Catelani, Ciani, Scarano, & Bacioccola, 2010), (Jones, Gray, Gold, & Jones, 2010), (Clarke & O'Connor, 2012)). The importance of the software development process in the achievement of the aforementioned goals has been emphasised ((Royce, 1970), (Boehm, 1988), (Munassar & Govardhan, 2010)).

Software testing plays an essential role as part of the software development process ((En-Nouaary, 1998), (Zheng, Alager, & Ormandjieva, 2008), (Holzworth, Huth, & deVoil, 2011), (Khan & Khan, 2014)). Khan and Khan have highlighted the importance of testing in enabling the validation of requirements. Software testing, a dynamic approach to software verification and validation, is not a unique tool in this respect, in fact many static methods have also been shown to be beneficial in helping to ensure the quality of software e.g. software inspections, automated source code analysis, and formal verification (Delahaye, Kosmatov, & Signoles, 2013). However, static methods, such as those mentioned previously, are performed against non-operational software, and cannot demonstrate whether the software is operationally useful. Software testing is described as an important method for validating software usefulness, and checking software quality characteristics, such as functionality and reliability (Holzworth, Huth, & deVoil, 2011). In support of this argument, En-Nouarry (1998), in reference to static techniques such as system specification verification, have stated that such methods do not guarantee the correctness of system implementations, and that testing is an important activity in this regard, one which aims to ensure the quality of such implementations.

Just as software testing attempts to validate software characteristics such as functionality and reliability, there are other important characteristics such as software complexity, which have a direct effect on the ability to perform effective software testing, (Zheng, Alager, & Ormandjieva, 2008). Some authors have referred to complexity associated with the modification of software ((Perrow, 1984), (Brooks F. P., 1986), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007)), along with complexity associated with software tasks in general ((Brooks F. P., 1986), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007)). Espinosa et al. state that this complexity varies greatly depending on the characteristics of the software task itself, like size and structure, and on environmental conditions, such as team size and the geographical dispersion of teams.

Perrow (1984) has made reference to the inherent complexity associated with technological systems in general and the potential negative consequences of such complexity. Complexity is also stated as an inevitable consequence of some system designs in order to achieve the intended goals of the system. Other authors have referred to the inherent complexity associated specifically with software systems ((Mumford, 1983), (Brooks F. , 1995), (Lehman, 1996), (Lyytinen, Mathiassen, & Ropponen, 1998), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007), (de Silva & Balasubramaniam, 2012)). Lehman (1996) has made reference to the naturally increasing complexity associated with evolving software systems (E-Type systems), unless deliberate attempts are made to reduce such complexity ((Lehman, 1996), (de Silva & Balasubramaniam, 2012)). Providing further insight into the concept of inherent system complexity, Brooks (1986), in line with the thoughts of Aristotle, has made the distinction between essential complexity, and accidental complexity, associated with software engineering. Difficulties associated with the nature of software, have been referred to as essentially complex, whereas difficulties associated with software production, have been referred to as being accidentally complex.

Debbarma et al. (2011) have argued that there has been increasing complexity, along with increasing size and performance demands, of software systems. All of which has demanded more effective software testing. The difficulty of providing test coverage for large or complex systems, has similarly been highlighted by other authors ((Zheng, Alager, & Ormandjieva, 2008), (Lin, Chou, Lai, Huang, & Chung, 2012)). Myers

(1979) also made reference to the difficulty and complexity associated with providing adequate test coverage (as did Ferrer et al. (2013)), and indeed the impracticalities of providing complete test coverage for software systems in general. Other difficulties associated with the role of the software tester, have been highlighted by Loveland et al. (2005), who have inferred that the goals of software testers have changed, from not only ensuring that among the defects found, are all the defects that would disrupt real working environments, but to also validating other system characteristics through specific testing, such as performance, and system recovery testing. Andrade et al. (2013) has expressed the view, that there have indeed been advancements made regarding software testing, and that older testing techniques, as devised by Myers (1979), have been added to by new testing models, such as model-based testing, and agile testing. As a result of this, new testing techniques have surfaced, such as machine learning techniques, adaptive random techniques etc. Notwithstanding these improvements, it is argued that such advancements, combined with the application of software to new domains and new development models, serve to make software testing an increasingly knowledge intensive and complex activity (Andrade, et al., 2013).

Rather than the identification of the difficulties and complexity which software testers face from a technological perspective, some authors have emphasised the importance of human factors, such as skill, experience, and management, in the achievement of software development goals ((Guinan et al., 1998), (Espinosa, 2007)), and their particular relevance in the achievement of software testing goals, (Martin, Rooksby, Rouncefield, & Sommerville, 2007). Guinan et al. have stated that the aforementioned factors, namely skill, experience, and management, are more effective enablers of software project success, than tools and methods. Faraj and Sproull (2000) and Ryan and O'Connor (2009) have also questioned the contribution of technological solutions to the performance of successful projects, instead highlighting the importance of human factors. Ryan and O'Connor (2009) and Dingsøyr and Šmite (2014) have referred to the importance of human factors such as effective plans, good communication, and clear goals, providing a clear link between the role of knowledge, and the success of software development teams. The increasingly important role of knowledge in the software development process has also been emphasised by Rus et al. (2001), who have stated that it is necessary to leverage individual knowledge at a project and organisational level, so as to ensure optimal software development.

A link between knowledge of the individual and practical intelligence, is described by Wagner & Sternberg (1985), who stated that formal knowledge, tacit knowledge, and general aptitude, are all important elements of practical intelligence. Zack et al. (2009) referred to knowledge as being an organisations key resource, directly affecting organisational performance, and thus organisational financial performance. The importance of the management of knowledge both from a qualitative point of view, and from a quantitative perspective, has been emphasised. A distinction has been made by numerous authors between explicit knowledge and tacit knowledge ((Nonaka & Takeuchi, 1995), (Zack, McKeen, & Singh, 2009), (Holste & Fields, 2010)). Explicit knowledge has been described as knowledge which can be easily codified. In the case of a reliance on explicit knowledge, it is suggested that a documented approach to knowledge transfer makes more sense. Tacit knowledge is described as difficult to articulate in writing, and is normally acquired through personal experience (Joia & Lemos, 2010). Examples of such knowledge are given as scientific expertise, operational know-how, and technological expertise. The transfer of tacit knowledge is described as being best facilitated through person to person contact. The importance of both explicit knowledge and tacit knowledge has been emphasised by the aforementioned authors.

The importance of the role of knowledge to software development team performance has been emphasised by Chau and Maurer (2004), Turk et al. (2005), Joia and Lemos (2010), and Nidhra et al. (2013). The transfer of knowledge within software teams must be enabled, because it is unlikely that all members of a software team will possess all of the knowledge required, for all software development activities, (Chau & Maurer, 2004). It is suggested that effective communication between software development team members, facilitates the transfer of knowledge. Knowledge transfer and knowledge acquisition is something which Espinosa et al. (2007) has explored, as part of their investigation into the relationship between team and task familiarity, complexity, and the overall effect on team performance regarding virtual or geographically dispersed software development teams.

The following sections of this chapter provide the rationale for this study, the research objective and research hypotheses, with the final section of this chapter finishing with an overview of each chapter of this research.

1.1 Rationale for Study

System test has been identified as an important part of the software development process ((Eickelmann & Richardson, 1996), (Cai & Card, 2008), (Desai & Shah, 2011), (Kochhar, Bissyand, Lo, & Jiang, 2013)). The impact of complexity on software development processes, and relationship between complexity and knowledge transfer has been referred to by numerous authors ((Chau, Maurer, & Melnik, 2003), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007), (Ryan & O'Connor, 2009), (Pee, Kankanhalli, & Kim, 2010), (Staats, Valentine, & Edmondson, 2010), (Lu, Xiang, & Wang, 2011), (Wang, Huang, & Yang, 2012)). Espinosa et al. (2007) have stated that additional research of software development work environments is necessary, to help understand how to deal with the varying complexities which increasingly characterize software development environments. Fundamental aspects of development processes, which are common across different approaches to software development, have been highlighted by Huo et al. (2004):

1. *Software specification and design.*
2. *Software implementation.*
3. *Software verification and validation.*

Andrade et al. (2013) referred to the increasing complexity associated with software testing tasks, as an important aspect of software verification and validation. Their research is focussed on the verification of complete software systems, as carried out through system testing, carried out by an independent test team. This is as distinct from a more granular approach to software testing, which may be carried out through module or unit testing. The use of an independent test team has been endorsed by Talby et al. (2006), who have stated that independent testers allow a more comprehensive test coverage, especially in the case of complex development projects. The primary activities associated with software testing, have been identified by Eickelmann & Richardson (1996), and Desai and Shah (2011), as relating to:

- *Test planning.*
- *Test development.*
- *Test execution.*

- *Test failure analysis.*
- *Test measurement.*
- *Test management.*

Desai and Shah (2011) also emphasise the role which tacit knowledge plays in software testing. A general case for more research into the role of knowledge, as it relates to the software development process, has been called for by Herbsleb (2007), who have highlighted concerns regarding the general lack of research in the area of software development. In his paper on socio-technical coordination, it is claimed that many authors have applied plausible rules of thumb, to answer questions such as what development practices are most applicable under what circumstances. Some have held the view that this is due to the general lack of empirical evidence available, relating to the stated benefits of software development methodologies ((Mitchell & Seaman, 2009), (Lee & Xia, 2010)). Due to this lack of empirical evidence surrounding the benefits of particular development approaches, it can be difficult to identify suitable characteristics of particular methodologies, which are backed by empirical, rather than anecdotal evidence. Cataldo and Ehrlich (2012) have referred to the lack of existing research, which examines the communication structures facilitating the transfer of knowledge, something which is considered key in software development processes, and also the overall achievement of software development goals such as productivity or quality.

The following section details the research objective and research hypotheses.

1.2 Research Objective and Research Hypotheses

A primary objective of this research is to add to or extend empirical evidence relating to the role which tacit knowledge plays in software system test complexity. The case for research in the area of knowledge, including tacit knowledge, and the role which it plays in software development processes, has been made by Ryan and O'Connor (2009), Von Krogh (2012), and Dingsøyr and Šmite (2014), who have emphasised the need for a greater understanding of this topic.

With the aforementioned research objective in mind, the first hypothesis takes account of the views of McKeen et al. (1994), Huo et al. (2004), Debbarma, et al. (2011) and Li, et al. (2011), relating to task complexity, and the views of others relating to the significance of inherent complexity, ((Mumford, 1983), (Brooks F., 1995), (Lehman, 1996), (Lyytinen, Mathiassen, & Ropponen, 1998), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007), (de Silva & Balasubramaniam, 2012)). Also acknowledged are the views of Ryan and O'Connor (2009), Desai and Shah (2011), Nonaka and Von Krogh (2009), and Hedesstrom (2000) regarding tacit knowledge. This hypothesis puts forward the premise that system testing is affected by complexity related to the system under test, and that most of such knowledge does not lend itself to being made explicit. In addition to the aforementioned views, the second hypothesis takes account of the work of authors such as Andrade et al. (2013), and Brooks (1986), with a distinction being made between essential complexity and accidental complexity associated with software engineering. The second hypothesis proposes that such a relationship exists between complexity associated with system test testing, and the system under test.

1. The process of system testing (comprising of *test case planning*, *test case development*, *test case execution*, test case fault analysis, test case measurement, and *test case management*), is directly affected by complexity associated with the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit knowledge. It is also proposed that most of this tacit knowledge does not lend itself to being made explicit.

2. That the process of system testing (comprising of test case planning, *test case development*, *test case execution*, *test case fault analysis*, *test case measurement*, and *test case management*), is affected by other sources of complexity, independent of the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit knowledge. It is proposed that much of this tacit knowledge does indeed lend itself to being made explicit.

A complete summary of each chapter is provided as part of the next section.

1.3 Research Summary

Chapter two provides an overview of different approaches to software development, and the implications for software testing. As part of this discussion, the common fundamental aspects of all software development processes are discussed, in line with the views of Huo et al. (2004). These are highlighted as:

1. *Software specification and design:* The functionality and constraints associated with the software must be defined. This may take the form of requirements definition and software and system designs, or alternative approaches such as user stories, system metaphors, architectural spikes, and release planning.

2. *Software implementation:* In line with the requirements, goals and designs, the software must be produced. This can normally be a planned iterative development process, or a planned sequential development process.

3. *Software verification and validation:* The software should be validated to ensure it acts in accordance with customer requirements or standards. Code verification can take the form of static checks such as code reviews, inspections, and peer programming, or dynamic approaches such as software testing, taking the form of unit and system testing. Validation can take the form of customer feedback and acceptance testing.

The aforementioned fundamentals are determined by the software development methodology which is adopted, so a review of prominent approaches to software development has been carried out in chapter two. Rajagopalan (2014) have stated that concerns over quality and the future maintenance of software, led to the widespread adoption of Royce's waterfall model (Royce, 1970). According to Rajagopalan (2014), the perception that Royce was promoting the concept of inflexible partitioning as part of his model, was the primary driver for subsequent software development models. The necessity of a more flexible approach to software development and the emphasis of a "practice over process" approach is something which is emphasised, particularly by those who advocate a more agile approach to software development. Highsmith and Cockburn (2001), and Chau (2004), have expressed the views that changing customer

requirements should be embraced, and that models that enable such a rapid software change (similar to those advocated from an agile approach) are superior. The focus on the software development process characteristic of flexibility, particularly by agile development methodologies, has resulted in a concentration on certain aspects of software testing. Crispin and Gregory (2009) have referred to the emphasis on agile as being reflected in the associated software testing. Such testing is stated as being defined by the business experts' desired features and functionality, and not generally by tests which critique the product.

The following general stages of software testing were identified, as part of a discussion relating to the validation and verification of software. These stages are in line with the work of Eickelmann and Richardson (1996) (similar functions have been outlined by Desai and Shah (2011)):

1. ***Test Planning*** includes the development of a plan relating to test case development. This is described as including the foundations for the *test objectives,* encompassing features of the system to be tested, risk assessment issues, organizational training needs, required and available resources, comprehensive test strategy, resource and staffing requirements, roles and responsibility allocations, and overall schedule.

2. ***Test Development*** is essentially the development of a *test approach,* which includes the specification and implementation of a test configuration.

3. ***Test Execution*** includes the execution of the implemented source code, and recording of execution details. The output of this stage includes test output results, test execution details, and test status.

4. ***Test Failure Analysis*** includes behavior verification and documentation, and an analysis as to the root cause for test execution failure.

5. ***Test Measurement*** is closely linked with test execution results, and test failure analysis. This stage encompasses test coverage measurement and test failure measurement.

6. ***Test Management*** relates to the management of test infrastructure and test resources. This includes management of test environment, including test environment state preservation.

Chapter three discusses the relationship between the task of system testing (as opposed to unit or integration testing), complexity, and the corresponding relationship to tacit knowledge. Regarding the relationship to complexity, a number of key perspectives are highlighted i.e. *inherent software complexity, software project complexity,* and s*oftware task complexity*. Subsequent sections of chapter three have made reference to the strong relationship between complexity associated with aspects of the software development process, and knowledge, from a both a general perspective ((Staats, Valentine, & Edmondson, 2010), (Wang, Huang, & Yang, 2012)), and specifically from a geographically distributed development team perspective (Espinosa, Slaughter, Kraut, & Herbsleb, 2007). Authors such as the aforementioned and Chau et al. (2003), and Cataldo and Ehrlich (2012), have discussed the topic of knowledge relating to software development in great detail. However the case for research on the topic of knowledge, including tacit knowledge (as distinct from explicit knowledge), and the role which it plays in software development processes has been made by Ryan and O'Connor (2009), and Dingsøyr and Šmite (2014), who have emphasised the need for a greater understanding of this particular topic. Whereas explicit knowledge is stated as having universal character, employed consciously, and not tied to any particular context, tacit knowledge is described as being tied to actions, procedures, commitments, ideals, values and emotions, with a strong relationship to past experiences, true beliefs, and the actions of intuition, and implicit rules of thumb (Nonaka & Von Krogh, 2009). Of interest in this research case, is knowledge as it applies to the task of system testing, such as discussed by Desai and Shah (2011) and Mantyla and Lassenius (2012). This is adopting a more specific view of the subject, taken by Staats et al. (2010), who also discussed the relationship between task complexity and tacit knowledge. The aforementioned discussions provide us with the two primary considerations for this research:

1. *Complexity associated with the task of system testing.*
2. *The relationship between system test complexity and tacit knowledge.*

The first consideration of this research, detailed above i.e. *complexity associated with the task of system testing*, has been broken down further in keeping with the views of McKeen et al. (1994), and Brooks(1995), with a distinction being made between task complexity and system complexity. Thus task related complexity has been viewed from the following perspectives:

1. *Complexity associated with the system under test.*
2. *Complexity associated with the process of software development*.

The concept of tacit knowledge, an important aspect of the second research consideration detailed above i.e. *the relationship between system test complexity and tacit knowledge*, is discussed as part of section 3.4 in chapter three. The views of Nonaka and Von Krogh (2009), who have asked a number of questions relating to organisational knowledge creation and the relationship between explicit knowledge and tacit knowledge, are highlighted. Explicit knowledge and tacit knowledge are described as both being conceptually distinguishable along a continuum, a view acknowledged by Hedesstrom (2000), and supported by Collins and Kusch (1998), and Ribeiro and Collins (2007). Tacit knowledge is described as being accessible through consciousness, if it leans towards the explicit side of the continuum. However, most of the knowledge relating to skills, due to their embodiment, is described as being inaccessible through consciousness. This point is echoed by Hedesstrom (2000), who makes an attempt at categorising the views of Nonaka and Von Krogh (2009), Polanyi (1966), and Tsoukas (2002). He states that the views of the aforementioned authors can be encapsulated, by distinguishing between:

- *Tacit knowledge which has not been formalised because of cost or time limitations.*
- *Tacit knowledge which has not been formalised because of the form of the knowledge, such as embodied knowledge.*

Hedesstrom (2000) has also made reference to the acceptance amongst a growing number of authors, regarding the clear distinction between tacit knowledge and explicit knowledge.

Chapter four outlines a research model and methodology. As a result of the discussions which were carried out in chapter two, and chapter three, the following two hypotheses were put forward for further investigation:

1. The process of system testing (comprising of *test case planning*, *test case development*, *test case execution*, test case fault analysis, test case measurement, and *test case management*), is directly affected by complexity associated with the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit knowledge. It is also proposed that most of this tacit knowledge does not lend itself to being made explicit.

2. That the process of system testing (comprising of test case planning, *test case development*, *test case execution*, *test case fault analysis*, *test case measurement*, and *test case management*), is affected by other sources of complexity, independent of the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit knowledge. It is proposed that much of this tacit knowledge does indeed lend itself to being made explicit.

The method of data collection which was proposed was a series of interviews, a similar technique to that conducted by Ryan and O'Connor (2009). Flanagan's critical incident technique was employed, a technique which has been used by Kaplan and Duchon (1988), delivered via a series of open questions. Four organisations were selected for participation, with the corresponding test teams responsible for testing 10 different systems in total. A preference was expressed that face to face interviews be facilitated, where feasible. The test teams varied in team sizes, from four testers to ten testers, with all teams operating with some level of geographically dispersion between team members. Tester experience of the participants varied from 1 years' experience to greater than 20 years' experience. There was also a variation in the employed development methodology, across the different development environments involved, with some teams operating in what was considered a traditional development environment, and some operating in an agile development environment.

The importance of the identification of research aims, via pertinent research questions, has been highlighted by Fitzgerald et al. (2008). To aid the identification of system test complexity, questions which were presented to selected participants, have been detailed in section 4.2.3. The selected questions have been based on the previous work of numerous authors, detailed in chapters two, three, and four, some of whose views have been discussed in brief, in the previous section. The following section provides an overview of conclusions which can be drawn from analysis of the research data.

1.4 Research Conclusion

A primary objective of this research is to add to or extend empirical evidence relating to the role which tacit knowledge plays in software system test complexity. Chapter 5 details the coding and analysis of collected data relating to the proposed hypotheses. In line with these hypotheses, a distinction was made between complexity and tacit knowledge associated with the *system under test*, and complexity and tacit knowledge associated with the wider *process of system testing*. The process of system testing has been defined to include resource considerations and management, as well as complexity and tacit knowledge associated with the test environment, and considerations relating to the final system deployment.

Observations are also made as part of chapter five. In keeping with the research hypotheses, figure 1.1 provides a synopsis of the system test activities which have been observed as having a positive relationship between complexity and tacit knowledge, from both a *system under test*, and a *wider system process* perspective.

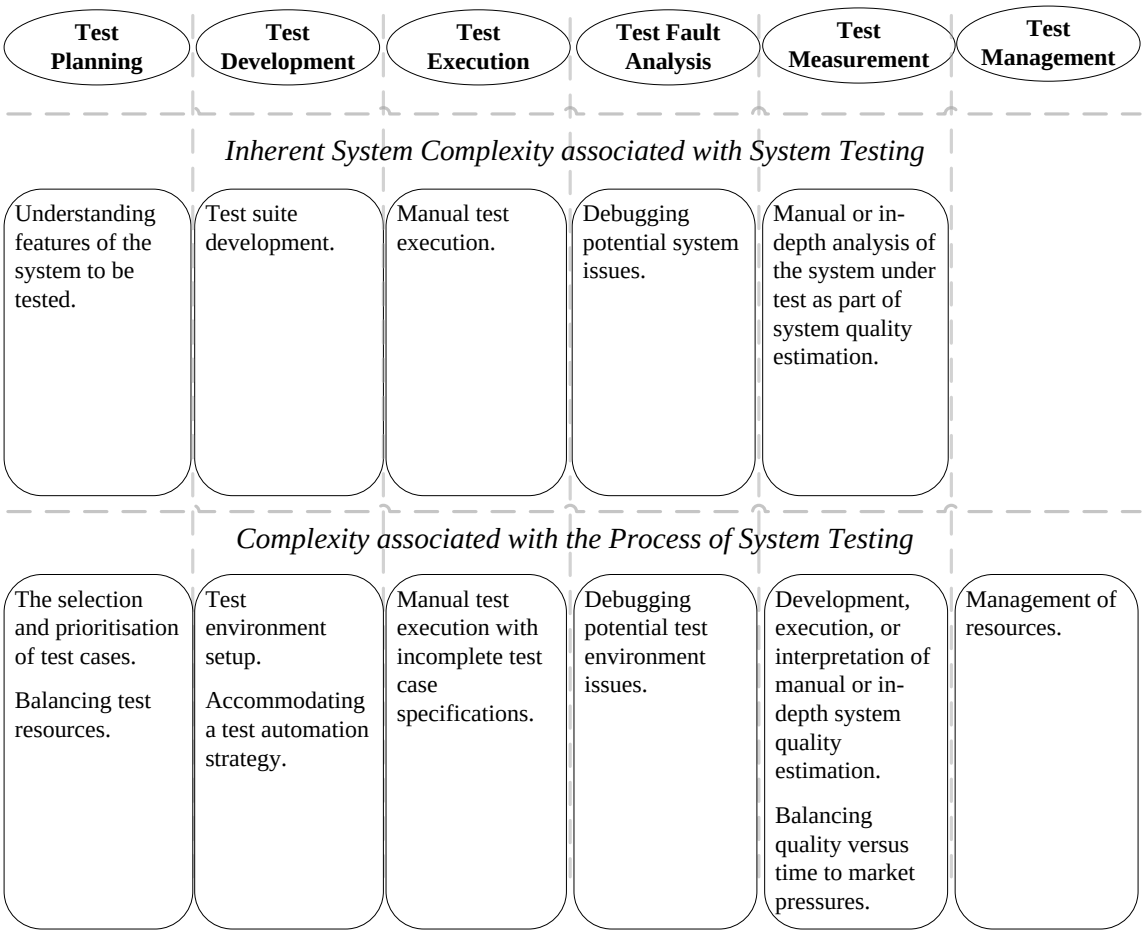


Figure 1.1: Sources of Complexity with a Direct Relationship to Tacit Knowledge.

Figure 1.1 is discussed in detail in sections 5.1 and 5.2. As part of the research activities, actions which can have a positive effect on the reduction of system test complexity were also identified in chapter five. These are discussed in brief in the following section.

Actions which have been proposed to reduce the effects of complexity

A number of actions were highlighted as part of section 5.4 relating to both the transfer of explicit knowledge and tacit knowledge. The source of such knowledge can be categorised as being *test team* related, *development team* related, or *application or support team* related. A model of the proposed actions identified as part of section 5.4, is detailed in figure 1.2.

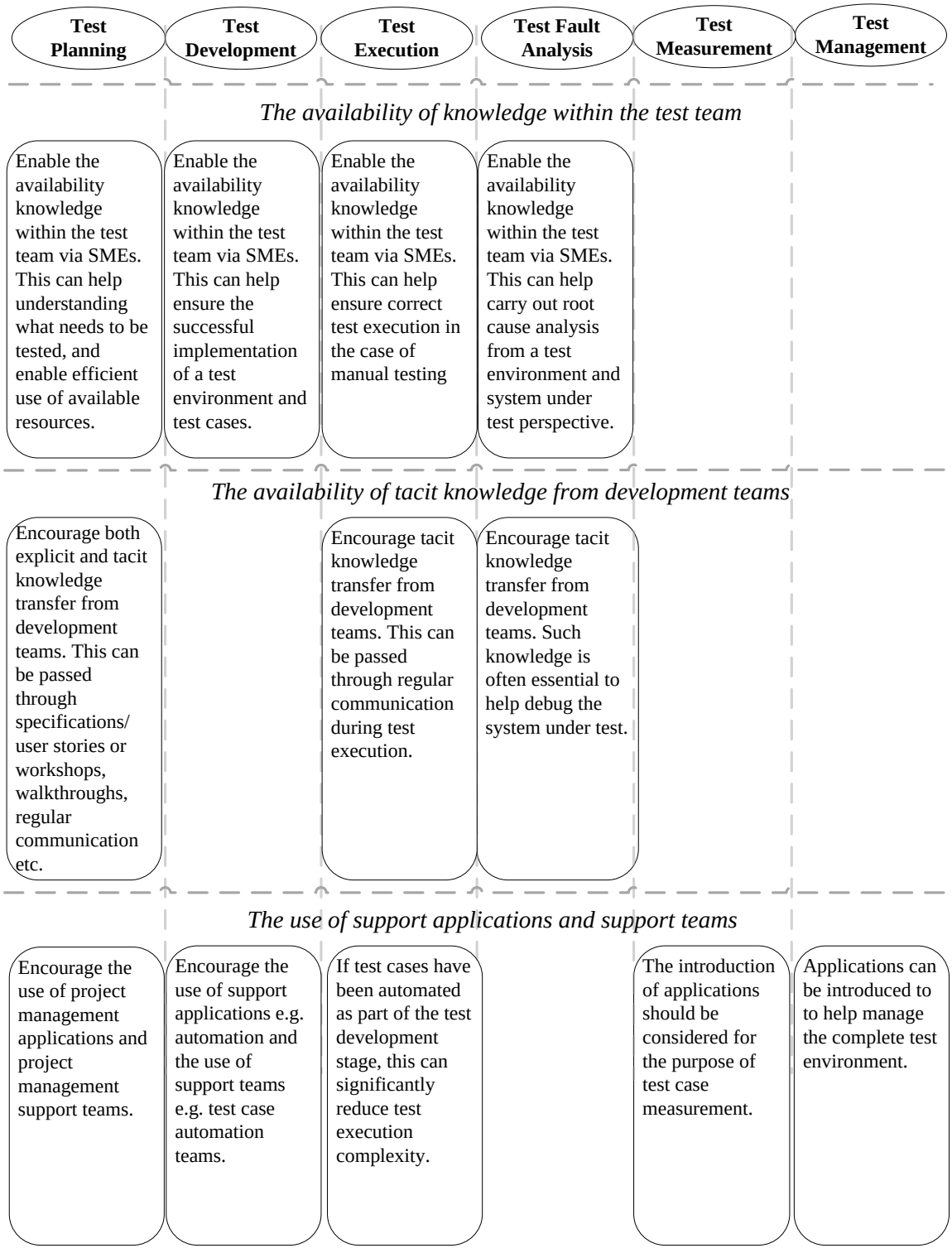


Figure 1.2: Recommended Actions Associated with Complexity.

Applying the views of Hedesstrom (2000), there is at least some applicable knowledge identified, which falls into the category of knowledge which could be made explicit due to time or cost limitations i.e. *explicit system knowledge* e.g. specifications etc. and

knowledge made explicit through the *use of support applications*. Contrary to this, knowledge has also been identified relating *Subject Matter Experts (SMEs), test team members* and *development team members*, of which some at least, falls into the category of knowledge which has not been formalised because of the form of such knowledge. A complete discussion on this has taken place as part of chapters 5 and 6. The following section outlines implications for the development process, as a result of this research.

Research Implications

This research has identified the importance of the availability of both explicit knowledge and tacit knowledge, relating to both the *system under test*, and associated with the wider *process of system testing*. A certain amount of knowledge relating to the process of system testing, lends itself to being made explicit, whether through the use of applications, such as project management, automation, or test measurement applications, or through system related specifications, user stories etc. Benefits associated with enabling the availability of tacit knowledge via appropriate people have been identified in both the case of complexity related to the *system under test*, and in the case of complexity associated with the *process of system testing*. Such people may be test team accessible SMEs, or development team members.

While the importance of explicit knowledge has been reinforced by this research, there has been a lack of evidence to suggest that the availability of tacit knowledge to a test team is of any less importance to the process of system testing, when operating in a traditional software development environment. To cater for the availability of tacit knowledge relating to the system under test, and indeed both explicit and tacit knowledge required by system testing in general, an appropriate knowledge management structure needs to be in place. This would appear to be required, irrespective of the employed development methodology.

The next chapter introduces the concept of system testing and the role which it plays in the software development process.

2 A Review of Software Development and the Role of System Testing

In a bid to provide a basis for addressing the primary research question concerning *the relationship between system test complexity and tacit knowledge*, the objective of the literature review is to primarily focus on the following:

- *The software development process and the role of software testing.*
- *Types of complexity which can potentially have an impact on the task of software testing.*
- *The importance of tacit knowledge to the development process and in particular the importance of tacit knowledge to software testing.*

Development methodologies relating to both traditional development and agile development are discussed in this chapter. The purpose of this is to investigate common relationships which exist between software development and software system testing, and to give an appreciation of the common environments in which system testing operates. The first section of the next chapter deals with the types of complexity which can potentially have an impact on software testing, starting with a broader discussion on the sources of complexity in the software development process, i.e. software complexity, project complexity, with subsequent sections concentrating on the role of complexity as they apply to the task of software system testing.

The second section of next chapter covers literature associated with tacit knowledge and the importance of tacit knowledge to the software development process, and in particular, the importance of tacit knowledge to software testing. Literature associated with knowledge types are discussed, along with the concept of knowledge conversion, and the importance of knowledge transfer. The role of tacit knowledge in software development is discussed, and its importance to system testing made evident. Numerous authors have referred to the importance of software development methodologies to the software project goals of software quality, the cost of software development, and the speed of software development, albeit with varying emphasis being placed on some goals rather than others, depending on project and organisational priorities ((Huo, Verner, Zhiu, & Bahar, 2004), (Liu, Chen, Chan, & Lie, 2008), (Mitchell & Seaman, 2009), (Cataldo & Ehrlich, 2012)). In the case of Huo et al.

(2004) and Mitchell and Seaman (2009) the important role of software testing to software development is something which has also been highlighted.

Development methodologies are described as having a direct influence on software testing. They dictate the work environment in which testing operates, including the pressures, opportunities and ultimately the role of software testing. The aforementioned authors give examples of different flavours of development methodologies, detailing the implications of each method on cost, quality, and time to market. Although other development methodologies are discussed in this review, in line with the views of Mitchell and Seaman (2009) and Crispin and Gregory (2009), this review will focus on what is generally considered to be the two main categories of software development:

1. *Traditional* or *plan driven* software development, focussing on the waterfall approach to software development.
2. *Iterative*, also encompassing *incremental* approaches to software development.

It is important to provide an overview of the characteristics of the main development methodologies because, as stated by Sommerville (2007), there is no ideal development process, and many organisations have developed their own approach to software development, often in an effort to exploit the capabilities of the people in an organisation. It is also stated that software development processes are commonly developed in line with the key characteristics of the system to be developed, and the overall project goals. In the case of critical systems or geographically dispersed development teams, a more structured development process is often required (this view is endorsed by Turk et al. (2005) and Dingsøyr and Šmite (2014), whereas in the case of business systems, with rapidly changing requirements, it is common that small co-located development teams, and a flexible, agile process is likely to be more effective (Dingsøyr & Šmite, 2014). In line with the aforementioned views, the following sections identify the main characteristics of common development approaches, starting with what are commonly described as, the *traditional* software development approaches, (Huo, Verner, Zhiu, & Bahar, 2004), (Rajagopalan, 2014).

2.1 Traditional Software Development

In the 1980s and early 1990s, there was widespread view that the best way to achieve software was through a combination of careful project planning, quality assurance, the application of analysis and design techniques, and strict software development processes (Tsui & Karam, 2007). Significant time and effort has went into refining these original development techniques, and techniques such as the *waterfall approach* (the most prominent of these *traditional* techniques) are stated as having reached a mature and stable state, having been applied to both large and small development projects, (Huo, Verner, Zhiu, & Bahar, 2004). Such development approaches are stated as facilitating knowledge sharing through explicit and extensive documentation, (Chau, Maurer, & Melnik, 2003). Extensive documentation practices are also stated as enabling the evaluation of an adherence to processes and plans, as well as supporting quality improvement initiatives and satisfying legal regulations.

Developed by Royce (1970), the waterfall model has been described as taking both a linear and sequential approach, each phase depending on the preceding phase completing before the next begins, see fig 1. Each phase contributes key deliverables for the next. Mitchell and Seaman (2009) have described this model as the oldest and still most widely practised of development models. Rajagopalan (2014) stated that concerns over quality and the future maintenance of software, led to the widespread adoption of Royce's model. This resulted in the introduction of a formal requirements stage in the development process. It is also stated that this model provides important feedback loops between stages of development as well as guidelines to confine feedback to successive stages, in an effort to reduce development costs. Another important aspect of the waterfall model was the introduction of prototyping. This highlighted the benefits associated with the production of software models as early in the development process as possible. Such practices enabled earlier, more comprehensive validation of software designs.

Royce (1970) saw the dependency between development stages as a potential risk of his model. A prime example of this is given as the test stage, which validates important elements of the software. This stage is completed at the end of the process, and as such may highlight not just coding issues but program design issues, which could

potentially cause a rework of the program design. Similarly, it was highlighted that issues identified at the program design change, could cause a rework of the software requirements, and the subsequent analysis stage. Another disadvantage of the model is that it is conceded that additional steps to software project analysis and software coding, do not essentially add value to the software product, drive up costs, and are generally not desirable for development teams, because of the lack of creativity involved. It is stated however, that without the additional steps to analysis and coding, namely requirements, design and testing, that larger software projects are "doomed to failure" with cost overruns, quality issues, and development delays inevitable.

The Waterfall Approach to Software Development

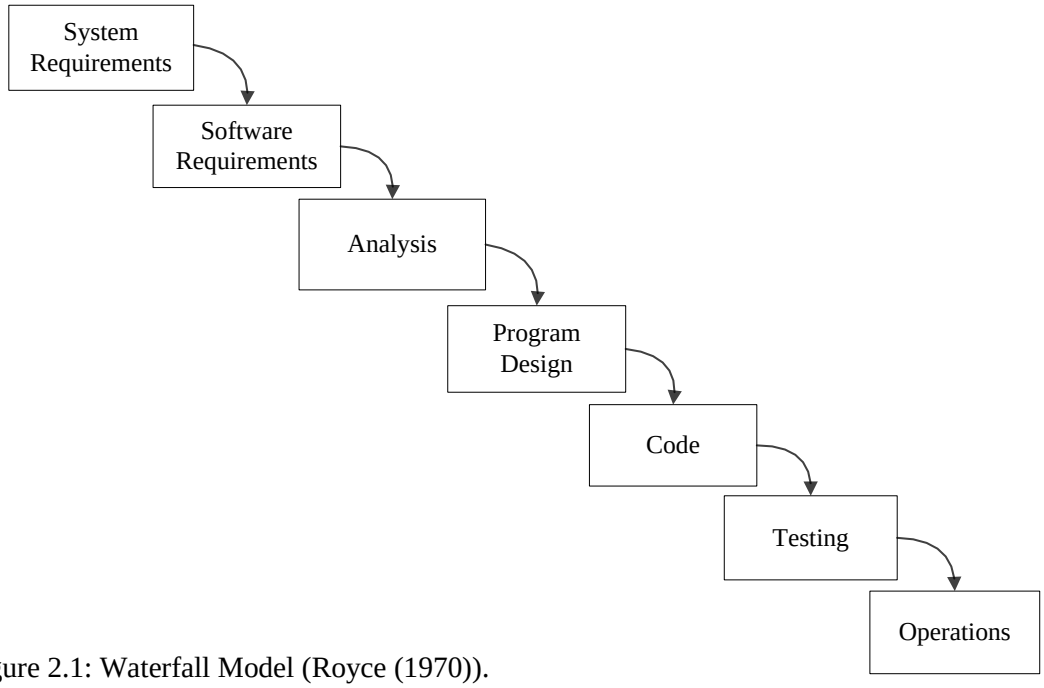


Figure 2.1: Waterfall Model (Royce (1970)).

Royce (1970) had five key steps which he believed were critical to eliminating development risk associated with the software development of large software projects (fig 2.1):

1. Ensure that the preliminary design is complete before the analysis begins, including system overview, defining data processing needs, applications interfaces, description of operating procedures and software performance times. This step is seen as key to avoidance of analysis issues at a later stage.
2. Ensure that all documentation is both current and complete. This was described as critical by Royce and includes document such as software requirements,

preliminary design specifications, interface design specifications, final design specifications, test plans, and operating procedures.

3. Contrary to the views of authors such as Chau el al. (2004) and Crispin and Gregory (2009), Royce actually did promote an iterative development model of sorts. He proposed that one should prepare to do a development job twice. The benefits of a preliminary model or prototype of the software product were seen as extremely beneficial by Royce in getting valuable customer feedback.
4. Plan, control and monitor software testing. Royce saw this as the biggest risk in terms of costs and overruns. He states that a number of elements are key to minimising the time spent on the test phase and to a successful test phase execution.
    - Visual code inspections should be carried out in advance of testing,
    - Every path in the software should be executed at least once as part of testing.
    - Testing should be carried out by an independent specialised test team.
5. Involving the customer as much as possible on a continual basis through the development process at stages such as requirements, software reviews and software acceptance, this is seen as key to successful project development. McCracken and Jackson (1982) have referred to the limited customer or end user involvement of traditional life cycle concepts but this would appear to be at odds with this key step which specifically highlights the benefits of customer involvement in validating requirements, design and functionality.

Checkpoint reviews are suggested to be carried out throughout the process. This enables progress assessment to be made against entry and exit criteria, in order to determine readiness for the next phase. The test phase is described as incorporating unit testing, functional testing, system testing, performance testing, and integration testing.

The sequential nature of this development approach has been referred to by Chau et al. (2004) and Crispin and Gregory (2009). Such methods involve the planning of the entire software development cycle with no formal plan for potentially unavoidable iterative development. Thus characteristics of the model encouraging sequential development were not perceived to be suitable in all circumstances. The inflexible partitioning of projects into distinct stages of development has also been referred to by

others (Huo, Verner, Zhiu, & Bahar, 2004). Huo et al. (2004) have also made reference to the impact of this inflexible partitioning, stating that in practice this has the potential to cause delays and cost overruns in the face of customer requirement changes. Although the concept of Royce advocating a concept of inflexible partitioning as part of his model, is something which is stated as having been misinterpreted from the original work of Royce (Rajagopalan, 2014), it is conceded that Traditional/Tayloristic/Plan-driven methods are more likely to encourage the adoption of a non-accommodating stance when requirements changes are suggested, thus leading to a higher probability of schedule and cost overruns. This view has been endorsed by Boehm (2002), who stated that a major contributor to this is the fact that testing is confined to final stages of development, and therefore any major issues identified are more likely to be subject to delays and inevitably cost overruns. Commitments made at early stages in the process have proved problematic in the face of changing customer requirements.

Davis et al. (1988) have referred to the benefits of the waterfall model in encouraging the specification of requirements and designs, enabling project management, the specification of tests. The structured approach also has benefits for future system modifications, should they be necessary, and enables knowledge transfer of explicit knowledge, something which is very beneficial in the case of distributed work teams, (Ramesh, Cao, Mohan, & Xu, 2006). In the case of the waterfall model, the method of knowledge transfer relating to the software development is clear. It consists of explicit, documented knowledge, being produced in the form of detailed specifications, which can then be interpreted by the test team and used in the development of test plans. According to Ramesh et al. (2006), this explicit approach to documentation, has distinct benefits in the case of distributed work teams.

It has been claimed that contrary to the original views of Royce (1970), that software development literature is rich with references to the misconception that Royce proposed a linear structure to software development, (Rajagopalan, 2014). As a result of such misconceptions, the view has been expressed that application of rigid processes, such as those detailed by the waterfall approach, are not suitable for application as part of the development of every software development project, a point which is referred to by Chau et al. (2003), and Huo (2004). Rajagopalan stated that

such concerns led to the introduction of iterative and incremental models,which are discussed in the following section.

2.2 Incremental and Iterative Software Development Models

Tarhan et al. (2014) have stated that low success rates of software projects during the 1990s (reported as being at 32% success rate (Standish Inc., 2009)) related to the application of Traditional software development models, led to the introduction of incrementally based Agile approaches to software development. Even prior to this, concerns had led to the introduction of other incremental and iterative development models, such as *the evolutionary model* ((McCracken & Jackson, 1982), (Perkusich, Soares, Almeida, & Perkusich, 2015)). The evolutionary model is described as an alternative to traditional, sequential, software development models, much in keeping with the *Spiral model*. (Boehm, 1988), which was also introduced in the 1980s. Both the *spiral model* and the *evolutionary model (EVO model)* adopt a more dynamic approach to testing, something which is discussed in more detail in the following sections.

2.2.1 The Evolutionary Model

McCracken and Jackson (1982) argued that it was not feasible for traditional software development models to be applied efficiently to all software projects. The authors make reference to the communication gap that commonly exists between end-users of software and software analysts, and put forward the notion that requirements cannot be stated in advance, because at such a stage the end-user does not fully appreciate the end requirements, not even in principal. The basis for this statement appears to be that requirements inevitably change throughout the development process, often due to a lack of realisation at the beginning by the end-user, as to what is actually feasible in terms of development. It is stated that any development environment must take account of the fact that the needs of the user, and the final working environment, is liable to change during the course of the development process.

Two suggestions are made, the first is to allow the product grow organically by way of models or prototypes, with analysts working hand in hand with the user, until the

acceptable product is developed. Under such a method the specification may never be written. The second suggestion is that an iterative process take place involving design, specifications, and implementation, with significant involvement again between the end user and the analysts. The difference between the suggestions is primarly that in the case of the second suggestion, interaction between the end user and the analyst, eventually results in the formulation of a design for implemention. Boehm (1988) had reservations regarding the proposal of this evolutionary model, stating that it was hard to distinguish between this and the old *code-and-fix* approach, whereby software implementation was the first step, and requirements, design, and test were thought about at latter stages of development. Boehm saw the following potential problems:

1. Issues involving *the integration of independently developed applications* which had not been properly planned.
2. Secondly, where *temporary work-arounds are deemed unchangegable* by the user after release of one iteration of development, this could make subsequent development more difficult.
3. Thirdly, in the case of *the software replacing a larger existing system*, it is stated that if a proper modular design does not exist, that it can often be difficult and complex process to provide a bridge between old software and the new software.

May and Zimmer (1996) developed their version of the *evolutionary model* (*EVO model*), and advocated the use of smaller iterative development cycles, which the authors maintain leads to better risk analysis and mitigation. The authors appear to have accounted for the lack of natural feedback associated with the waterfall method, something which the EVO model has included via feedback loops within the small waterfall cycles. Cycles associated with the EVO model, tend to last two to four weeks and include all aspects of design, code, and initial testing of a new version of software. Feedback from the prior cycle is evaluated during the execution of the next cycle. It is pointed out that in the case of complex software projects, smaller development cycles and smaller software components may not always be possible to adhere to. The basic principle is similar to that of the incremental model as discussed earlier, whereby software is released via code drops, each of which goes through design, development, and test prior to Beta testing. The difference is that within EVO, interim versions of the product are developed, and then provided to customers for feedback, whereas the

waterfall or similar traditional methods rely primarily on feedback from internal test groups, from a black box or white box perspective.

In the case of the model proposed by McCracken and Jackson (1982), it is difficult to see the role which an independent test team plays, if any at all. It would appear as though the operational usage by the customer is the actual execution of system verification and validation. Under the model proposed by May and Zimmer (1996), the relationship between development and test is clearly iterative, with formal interaction taking the form of specifications being incorporated into the model. This is in keeping with aspects of the waterfall approach. These specifications can then be utilised by the test team in the development of test specifications. Another model in response to the models proposed by Royce, and McCracken and Jackson, is the Spiral model, proposed by Boehm (1988). This model, which is discussed in the following section, attempted to retain the structure of the waterfall approach, while introducing incremental and iterative aspects to the software development process.

2.2.2 The Spiral model

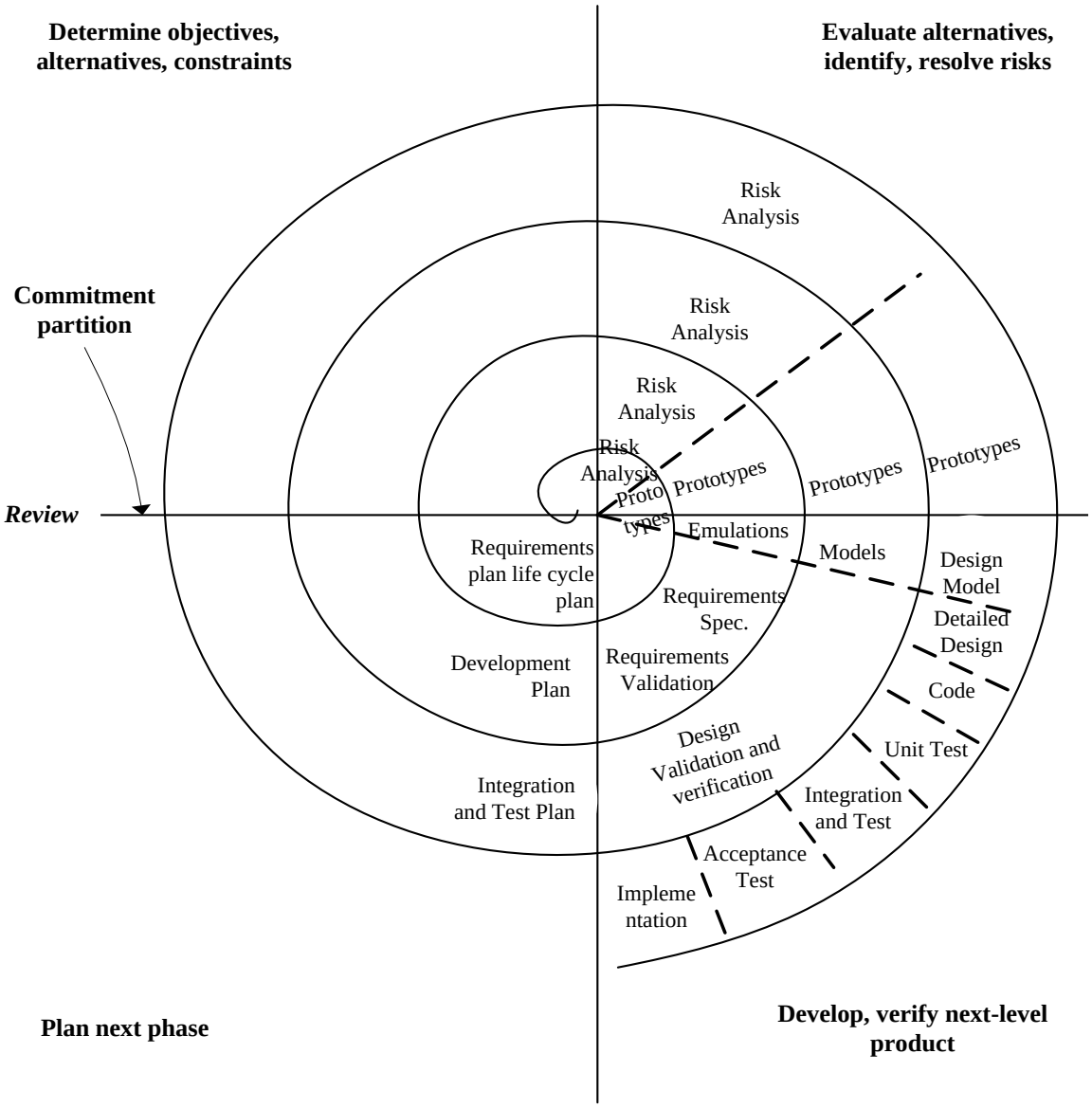
This model was proposed by Barry Boehm (1988) in response to concerns regarding the waterfall method (Royce, 1970), and the evolution model (McCracken & Jackson, 1982). The Spiral Model (fig 2.2) adopts three important principles from the waterfall approach:

1. *Feedback loops* between stages to avoid expensive rework at the end of the overall process.
2. *Introduction of prototyping* in the software life cycle as a means of validating requirements.
3. *A structured approach to requirements and design*, including associated documentation.

An iterative element was included in the model, in line with evolution proposals (McCracken & Jackson, 1982), as well as a risk analysis stage to allow the evaluation and resolution of project risks. The model is described as providing a cyclic approach

to incrementally developing software, while reducing the project risk as the project goes through cycles of development. As the software project journeys through the four quadrants associated with the model, it is incrementally developed. A cycle of the spiral typically begins with the evaluation of project objectives (functionality, ability to accommodate change etc.), the evaluation of alternative methods of implementation (based on alternative designs, outsourced, off the shelf software etc.), and consideration of the constraints imposed on the application (cost, schedule, interfaces etc.). The next step of the cycle is to evaluate the alternatives in terms of objectives and constraints and the identification of significant sources of project risk. The initial stage of development begins with the evolution of a prototype. As the project progresses there is an emphasis on the identification and evaluation of risk at each particular stage. Each loop in the spiral represents a phase of the software process i.e. the innermost may be concerned with system feasibility, the next with requirements, system design and so on. Each loop is split into four sections:

1. *Objective setting*: This relates to defining objectives for that phase of the project. Constraints on the process and the product are identified and a detailed management plan drawn up. Risks are identified and alternative plans may be drawn up based on identified risks.
2. *Risk assessment and reduction*: For each identified project risk, a detailed analysis is carried out. For example, if there is a risk that the requirements are inappropriate, then a prototype may be developed.
3. *Development and validation*: After risk evaluation, a development model is chosen. If user interface risks are prominent then an evolutionary prototyping model may be chosen. If multi-system integration is a main risk then the waterfall method of software development may be chosen.
4. *Planning*: The project is reviewed and a decision made as to whether to continue with a further loop of the spiral, if so then plans are drawn up for the next phase.



(Boehm, B. (1988)

Figure 2.1: The Spiral Model of Software Development.

Through prototyping, requirements definition, design, and implementation, each revolution independently examines the objectives, risks, implementation and planning of the phase that follows. This offers regular decision points for determining whether the software should continue to the next phase, stay in the current phase and continue efforts, or completely terminate the project. By evaluating the risks at each revolution of the spiral, improvements can be made to enhance software quality, or to bring the project back in line with original goals. Issues identified through analysis, can provide an opportunity to alter the development model to suit particular needs such as quality concerns. Such concerns could be addressed by scaling down development models for

instance. Continual analysis of risks, consistently provide the opportunity to assess whether focussed testing is unsuccessful in meeting quality concerns, and time to market challenges. The extension of the radial of the model is stated as representative of the cost increase as the project progresses, and the angular dimension of the model, represents the project progression.

The relationship between software development and software test is still linear with other validation and verification stages built in via the *simulations, models and benchmarks* stage to apply quality assurance at all of the development loops. There is a significant dependency on the ability to assess risk. Dependency also exists on the quality of simulations, prototypes and models and the quality of interaction with the end user to validate this phase of development. A risk also exists with the late application of testing, which as mentioned in the introduction, is considered to be the foremost method of software verification and validation. The next section discusses other incremental development models.

2.2.3 Other Incremental Models

The purpose of this review is solely to discuss models which add to the relationship between development and test, the role of system test, or knowledge transfer to system test. There have been other models which have been referred to as being prominent iterative models, such as the *cleanroom model* ((Mills, Dyer, & Linger, 1987), (Perkusich, Soares, Almeida, & Perkusich, 2015)). Along with *integration models*, and iterative versions of the waterfall model, or indeed hybrid models such as *Rational Unified Process (RUP)* model, which have been derived from work on the *Universal Modelling Language (UML),* and the associated *Unified Software Development Process*, (Rumbaugh & Jacobson, 1999). Such models are not deemed as adding additional value to this particular discussion and therefore are not discussed in any detail here. The waterfall model covers the static relationship between development and test, and the evolution models and the spiral model cover both incremental and iterative development, dealing with the repetitive, dynamic relationships which may exist between development and test. The iterative approach to testing would obviously

alleviate some of the potential test bottle necks associated with non-iterative models, such as the waterfall method.

The importance of incremental models, according to Sommerville (2007), is the separation of phases and workflows, and the recognition that deploying software in a user's environment must form part of the process. Phases are dynamic and have goals, whereas workflows are considered static and are technical activities, not necessarily associated with any particular phase, but which may be used throughout the development process in order to achieve the goals of each phase. There has been extensive work carried out on both traditional models and incremental models, but the view has been expressed that these methods did not go far enough to accommodate unstable or incomplete requirement changes, (Highsmith & Cockburn, 2001). The disadvantages associated with incremental models are investigated by Tarhan and Yilmaz (2014), who found that Agile development methodologies (designed to effectively and efficiently accommodate requirement changes) outperformed incremental models in terms of development productivity and quality. The development of a completely agile approach to software development is discussed in more detail in the next section.

2.2.4 The Conception of Agile Processes

During the 1990s, due to the need to reduce time-to-market, a major shift occurred away from sequential models towards agile (Perkusich, Soares, Almeida, & Perkusich, 2015). Highsmith and Cockburn (2001) and Tarhan and Yilmaz (2014) have stated that the strength of Agile development processes is the ability to accomodate unstable or incomplete requirements throughout the development and test phases, something which the waterfall and incremental model are not designed for. Such development enviroments enable software to be developed quickly to take advantage of new opportunities and to respond to competitive pressure, (Highsmith & Cockburn, 2001). Similar views have been echoed by Lee and Weidong (2010), who have stated that the primary objective of agile development approaches, is to place priority on the ability to effectivly respond to user requirement changes, something which the aforementioned

authors claim was not not sufficiently catered for with preceding iterative approaches for software development i.e. the spiral and EVO approach.

Agile development models provide the advantages of all iteratively developed software, which are *accelerated delivery of services* and *early user enagement with the system*. It is stated that agile development differs from traditional plan driven models, because of the focus on lean processes rather than detailed front-end plans and heavy documentation. Highsmith and Cockburn (2001) stated that there are common values which were identified via the *Manifesto for Agile Software Development.* These core values are centred around the notion that one must accept that requirement changes throughout a software development project are inevitable, and that the most sensible course of action is to attempt to reduce the costs associated with such changes. The core values identified are:

- *People not process*: The skills of the development team should be recognised and exploited and team member, should be free to develop their own methods of working, without prescriptive processes.
- *Software over documentation*: working software should be prioritised over the production of extensive documentation.
- *Customer involvement*: customer collaboration should be prioritised over contract negotiation.
- *Embrace change*: Expect the system requirements to change, so design the system to accommodate such change rather than following predetermined plans.

Lee and Weidong (2010) stated that core values and principles of agile development have primarily been derived from past experiences, supported by anecdotal evidence. In an attempt to redress that imbalance, the authors research the effects of two dimensions which they describe as key to agility:

- *Response extensiveness* - A software teams response extensiveness is defined as the proportion of various types of changing user requirements which a software team can accomodate. This, it is argued, indicates greater software development agility.

- *Response efficiency* – Software development team efficiency is defined as the minimal time, cost, personnel and resources that the team requires to respond to and incorporate a particular requirement change.

The first aspect of the research carried out by the authors was to investigate the relationship between these two dimensions. Another aspect of the research focusses on how the team characteristics of autonomy and diversity influence software development agility. Autonomy is described as the extent to which software teams are empowered with the authority and control to make decisions during the project. Team diversity is described as extent to which team members differ in terms of skills, expertise and work experience. Both team autonomy and team diversity are stated as being important principles of any Agile development team according to Larman (2004), and therefore consider this a valuable aspect of their research given the absence of any empirical research being carried out on this particular subject. The final aspect of the research was to examine how the two dimensions of software development agility, namely response extensiveness and response efficiency, affect development performance in terms of on-time completion, on-budget completion and software functionality.

Lee and Weidong (2010) found the following relationships:

- *Software teams inherently have a dynamic ability to evaluate and find the appropriate balance between software development agility and software development performance*. This is achieved through assessment of business impact, the impact on time, cost, scope, and the technical difficulty. Based on these assessments the appropriate response to user requirments changes is determined. It was found that response extensiveness has a positive effect only on software functionality, whereas response efficiency has a positive effect on time and budget completion, as well as software functionality. Agile practices which demand time and cost consideration when accepting requirement changes are useful for improving response efficiency. The non-significant effect of response extensiveness on time and budget concerns is explained by an extensive response which is dealt with later in the development cycle generally requiring substantial time, cost, and resources, whereas an extensive

response earlier in the development cycle can result in possible savings in development time and costs at later stages.

- *There is a tradeoff between the software teams response extensiveness and response efficiency*. Reasons for this include that extensive requirement changes were often found to require upper management signoff, due to significant business or project impact. It was also found that response efficiency can diminish through work overload, which also results in a lack of focus. It was found that managers found that they can strike the correct balance between efficiency and extensiveness, if user requirements are clearly specified and understood, and there exists effective management of time and cost. In contrast to the view of the author, a reduction in response efficiency due to a workload increase could also be due to a a natural increase in software complexity if the size of the software task was naturally increasing with the workload. Such as view would also be supported by Brooks (1986), who refers to the inherent complexity associated with software, which naturally increases as the size of the software task increases. Espinosa et al. (2007) and Perrow (1984) have also referred to the complexity associated with the modification of software, due to the tight coupling of software module interdependencies.
- *Team autonomy was found to have a positive effect mainly on response efficiency* because of the empowerment decisions made by team members. Autonomous teams tend to limit their response to changing requirements in order to meet project goals. This is in contrast to less autonomous teams whereby teams may have no choice but to attempt to implement requirements with little regard for project goals. This may also explain why autonomy may have a negative effect on team response extensiveness. The findings could be explained by the importance of knowledge transfer to teams, as referred to by other authors ((Chau & Maurer, 2004), (Ryan & O'Connor, 2009), (Cataldo & Ehrlich, 2012), (Dingsøyr & Šmite, 2014)). Chau and Maurer (2004) referred to the dependence of teams on knowledge and emphasise the importance of short communication chains for optimal transfer of such knowledge.
- *Team diversity was also found to improve response extensiveness* because it helps solve various problems effectively and helps in understanding a wider variety of requirements specifications, possibly due to a greater availability of expertise and skills. Diversity was also found to possibly have a negative effect

on response efficiency due to costly conflicts, and costly communications. Supporting the view that team diversity was found to improve response extensiveness, Chau and Maurer (2004) argued that it is unlikely that each team member will possess all of the skills necessary for a successful project implementation, therefore it would appear plausible that team diversity would have a positive affect. As mentioned in the previous section, the aforementioned authors also emphasise the importance of short communication chains for optimal transfer of knowledge, which may aid reponse efficiency.

With relevance to this particular research, Talby et al. (2006) have stated that in a traditional development environment that everyone is responsible for quality, but in an agile development environment, test becomes part of each team members work, including developers, business analysts and even customers. This makes agile methodology, a "test-driven development model", with software test acting as a key measure of both team and personal productivity. Tests are devised prior to development being completed, thus focussing on highlighting any software defects as early in the development cycle as possible. Crispin and Gregory (2009) have highlighted the negatives of the agile approach to testing, stating that under such an approach, the testing defined by the business experts' desired features and functionality, and not generally by tests which critique the product. Concerns over agile process are also raised by Turk et al. (2000) who stated that agile processes are designed to provide developers with an environment to develop software as fast as possible, which can also cause it's own efficiency problems. There is a risk that in the application of such approaches, that software development productivity can often take priority over software reuse. It is also stated that the agile development works well for small teams in close proximity with continuous access to end users, which is unfortuanely not always posible in larger organisations, (Ramesh, Cao, Mohan, & Xu, 2006). This has implications not only for the efficiency and effectiveness of development but also for software test, a point which is highlighted by Ramesh et al. (2006). Andrade et al. (2013) have referred to the complexity associated with testing, which has increasd along with the progress of development methodologies.

There has been some suggestion as to the superiority of extreme programming (XP), (Beck, 1999), (Beck, 2000). Others have maintained that insufficient research has been

carried out examining the key concepts and underlying principles of agile approaches to software development ((Baskerville, 2006), (Dyba & Dingsøyr, 2008), (Mitchell & Seaman, 2009), (Petersen & Wohlin, 2010)). Dyba and Dingsoyr (2008) accept the widespread use of agile development practices, but state that software development agility is difficult to achieve in practice, with key principles and benefits not based on scientific evidence. This view would appear to have been endorsed by Mitchell and Seaman (2009), who carried out a review of research, comparing the waterfall method of software development, against available research on a varierty of iterative and incremental development methodologies. The authors firstly found a lack of empirical evidence which actually compared the two perspectives and secondly, research which they found did not demonstrate any identifiable cost, development duration benefit, or quality differentiation, between the two perspectives. Some additional research in this area has been contributed by Tarhan and Yilmaz (2014). They have indeed found there to be an empirical advantage in the case of adopting an agile approach to software development, regarding software quality and development performance, but have expressed similar sentiment regarding the necessity for further research to be conducted.

Other agile approaches do exist, such as Scrum (Schwaber & Beedle, 2001), Crystal (Cockburn, 2001), Adaptive Software Development (Highsmith J. , 2000), DSDM (Dynamic Systems Development Method) (Stapleton, 1997), Feature Driven Development (Palmer & Felsing, 2002), but XP has been described as the most popular of agile methods ((Martin R. , 2003), (Tsui & Karam, 2007)). Given the similar underlying characteristics of the aforementioned agile appraoches to software development, and the popularity of XP, this is the only agile software development approach which is discussed here in any detail. All of the agile development models appear to have the common characteristics of:

1. *The processes of specification, design and implementation run concurrently*. There is no detailed system specification, and design documentation is minimised or generated automatically by the programming environment. Usually only the most important characterics of the system are defined as part of the user requirements document.

2. *The system is developed in a series of increments*. End-users and other system stakeholders are involved in specifying and evaluating each increment after which changes and new changes are proposed to be catered for via subsequent increments.
3. *System user interfaces are often developed using an interactive development*. This enables quick creation of interface design.

eXtreme Programming

eXtreme programming (XP) is used as an example of an agile development method because as previously mentioned, XP has been described as the most popular of the agile methods. XP was developed out of necessity for software development methodologies to embrace and deal with change efficiently throughout the software life cycle, rather than attempt to specify all requirements at the the beginning of a software lifecycle and discouraging changes at later stages, Highsmith and Cockburn (2001). Accepting that change is inevitable, XP attempts to deal with change efficiently, by validating work as soon as possible in the development process. The following steps are an attempt to reduce the cost of change whilst retaining quality:

- *Produce the first delivery in weeks*.
- *Invent simple solutions*, thus allowing easier evolution of software.
- *Improve design quality continually*. This is stated as helping to reduce the costs of the next story or iteration of development.
- *Test constantly and as early as possible* in order to keep development costs to a minimum.

XP, as with other Agile processes, is designed to enable swift reaction to changing customer requirements, (Paetsch, Eberlein, & Maurer, 2003). Critical to the development process is the formulation of user stories which provide a description of a particular feature aimed at providing business value to a customer. This process of detailing customer expectations through methods such as brainstorming and interview processes, is descibed as being based on the important characteristic of feedback, between the customer of the software and the developers. Contrary to the other development methodologies, whereby feedback is also a necessary characteristic to aid

error correction and design flaws, in the case of XP, feedback is used to actually create the design, and provide the development team with sufficient information to estimate development effort. This in turn enables the development of user stories, leading to explicit user requirments and expectations. As is the case with other agile development practices, XP makes extensive use of *test driven development*. Acceptance tests are defined by the customer against user stories. These tests are created up front, prior to implementation of the software they will run against. The purpose of this method, is so that the developer is constantly considering the tests which his software will have to pass. Talby et al. (2006) appear to disagree with the concept of developers detailing tests. They have referred to the benefits of the use of independent test professionals in writing such tests. Tests are batched together and each release of software must pass all defined tests.

A difference between traditional software development methods and XP, is that XP doesn't provide the requirements and design documents which traditional software development models demand. In keeping with other agile development methods, documentation is discouraged beyond what is necessary to implement the code correctly, with product and task knowledge becoming increasingly tacit, (Nerur, Mahapatra, & Mangalaraj, 2005). The aforementioned authors refer to the importance of the transfer of knowledge between team members, which could be facilitated by a continuous rotation of team membership, thus ensuring this knowledge is not monopolized by a few individuals. The importance of knowledge management in testing is emphasised by Andrade et al. (2013). In the absence of such measures, a lack of documentation may impede future modifications of software, particularly in the absence of the availability of the original developers, who may have moved on to other work after a project has completed. The dependence on tacit knowledge within agile teams instead of formal documented knowledge is something which has been highlighted ((Paetsch, Eberlein, & Maurer, 2003), (Turk, France, & Rumpe, 2005), (Dingsøyr & Šmite, 2014)). Paetsch et al. have stated that whilst traditional software development tend to err on the side of overdocumentation, agile approaches such as XP tend to underestimate the risks due to a lack of proper documentation which could serve to offset knowledge loss, due to the unavailability of the original developers.

This possible deficiency in necessary knowledge, has a potential impact on all aspects of the development process including system test ((Paetsch, Eberlein, & Maurer, 2003), (Chau, Maurer, & Melnik, 2003)). Through the development of customer driven acceptance tests, agile caters for functional requirements but quesions have been asked as to the ability of agile methodologies such as XP to handle non-functional requirements such as *maintainability*, *portability*, *safety* or *performance*. Other authors raise questions as to whether agile software development such as XP processes, are suited to a large complicated project, where documentation, strict quality control, and objectivity, are critical, (Sommerville I. , 2007).

In a comparison made by Huo et al. (2004), between waterfall and agile software development approaches from a software quality perspective, it was established that the development of code at an earlier stage in the development process, invites the application of quality assurance techniques at an earlier and continually throughout the cycle. Testing is stated as being integrated into the development phase, with early and continual customer releases bringing customer feedback for product validation and requirements verification. Huo et al. (2004) did not detail a specific role for a seperate software test team, instead highlighting the following aspects of quality assurance to be applied:

- The application of *test driven development (TDD)*, whereby developers create their tests prior to software implementation. This leads to a constant focus on customer requirements from the project outset with tests being designed in line with known requirements, and acknowledgement by development of tests the system will have to pass.
- The application of *static techniques* such as code inspections, pair programming, refactoring, collective code ownership (shared responsibility for all sections of code), and coding standards.
- Early product validation through early software releases allow acceptance testing and encourage continuous integration.

Contrary views to the model of test driven development were put forward by Talby et al. (2006), who carried out research of the application of professional testers in an agile development environment, associated with a large-scale project. Given the

increasing complexity associated with testing, a model of a professional test approach is also supported by Andrade et al. (2013). The complexity associated with the development of acceptance tests for such a project, required that professional testers be employed in order to achieve comprehensive test coverage. The use of an independent test team in an agile software development environment, is described as a common practise in larger software projects, and something which is also referred to by Striebeck (2005). In the case of the research carried out by both Striebeck and Talby et al., the authors found some evidence to suggest that when test was not closely integrated with development during the development process, but rather carried out in a more traditional manner i.e. testing subsequent to development freeze dates and testing in test scripts developed in accordance with development specifications, there were some mismatches found between the system specifications, and the software system, but relatively few bugs found with the actual software itself. This is explained as a result of relatively comprehensive unit testing being carried out by development prior to the test team receiving the software. In both of the aforementioned cases, when the test team is integrated with the development process, it was generally considered a more efficient and productive approach for the acceptance testing to pursue. In such as scenario, the test team works to define tests in paralell with developers during the software planning and implemention phase.

In both research cases, the development of an automated test suite was considered the more productive option. In the case of the research by Striebeck (2005), it was the actual developers who implemented the automated acceptance tests after they were defined in consultation with the test team, but in this case it was considered a more beneficial option, if the test team was closely integrated with the development team during implementation of the actual acceptance tests. The role of an independent test team in carrying out quality assurance, as previously referred to, is important aspect of this particular research due to the specific focus on the relationship between development teams and test teams. The following section provides an overview of development processes, as well as a more detailed overview of the role of software testing, it’s prominence as a quality assurance technique, and the characteristics which define it.

2.3 A Synthesis of Software Development Models

This section initially defines the role of software test as a quality assurance technique, and proceeds to detail the characteristics of software testing. It must be noted that in addition to the aforementioned software development approaches, of which software test plays a significant role, many organisations also employ software process assessment and standardisation models in an attempt to achieve quality, cost, or schedule goals, ((Tsui & Karam, 2007), (Liu, Chen, Chan, & Lie, 2008), (Perkusich, Soares, Almeida, & Perkusich, 2015)). Such process assessment and standardisation models have a direct effect on the role of software test within any development process, so merit some discussion. Tsui and Karam (2007) have made reference to both the Software Engineering Institute's Capability Maturity Model (CMM), and Capability Maturity Model Integrated (CMMI), which are described as frameworks used to help organisations define its level of software development maturity. Also referenced is the International Standards Organisation (ISO), which defines a series of software quality standards, such as the ISO 9000 series, including standards which can be applied to software activities. It is stated that software engineering development and support processes, have continued to be modified, improved, and invented through countless studies, experiments and implementations, to varying degrees of success and failure. Liu et al. (2008) stated that the goal of such standards is to aid organisations instil better controls, through structured activities during development of a software product or system.

The aforementioned authors investigated the relationship between the standardization of the software development process, software flexibility, and project performance. The importance of software systems to be flexible or easily modified, to enable the accommodation of new user requirements, is something which authors such as de Silva and Balasubramaniam (2012) have also referred to. Liu et al. (2008), through their investigation into whether software standardisation has a positive or negative relationship on software flexibility, and final project performance, found evidence of such a relationship. Therefore it is advised that software flexibility concerns should be considered in an effort to standardise software processes, because substantial parts of software process improvement frameworks, or the implementation of standards of practice, are biased towards discipline (control), rather than creativity. This is described as something which can have a negative impact on software flexibility, or

the degree to which the software can be maintained or changed. This view relating to the flexibility and modifiability of the software subject is supported by Jamwal (2010).

Alternatively, Kelly (2008) have made a distinction between the benefit of software standards, as applied to software destined for a safety critical role, and software projects which demand a more flexible and innovative approach to development. The view is expressed that software standards often struggle, with enabling the achievement of product integrity, which is key to all software systems. It is stated that software standards often find it difficult to bridge the gap, between obtaining the required goal of the software, and meeting the needs of the customer, and how to implement that correct usage at a coding level.

2.3.1 The Case for a Flexible Approach to Software Development

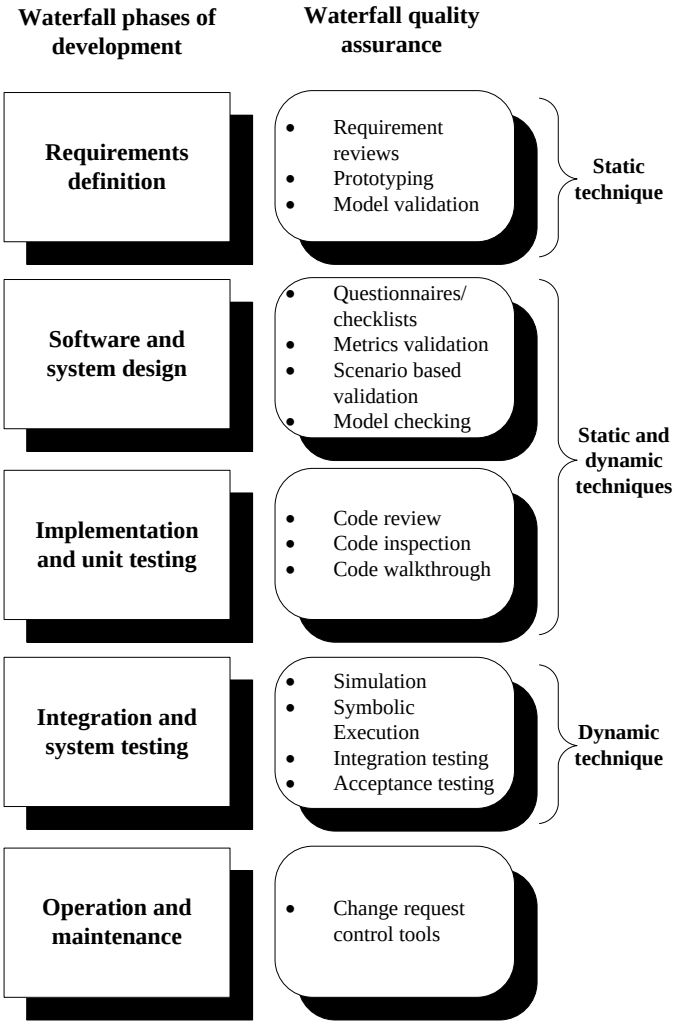
The necessity of a flexible approach to software development and the emphasis of a "practice over process" approach is something which is emphasised by those who advocate a more agile approach to software development. Highsmith and Cockburn (2001) and Chau (2004) have stated that changing customer requirements should be embraced, and that models that enable such a rapid software change (similar to those advocated from an agile approach) are superior. It should be noted however that Martin et al. (2007), and Mitchell and Seaman (2009) have cited the lack of empirical evidence to back up such claims, and Lee et al. (2006) have cautioned against the promotion of software development flexibility at the expense of explicit documented knowledge, particularly in the case of geographically distributed software development environments. Although Tarhan and Yilmaz (2014) do actually provide empirical evidence in support of an agile approach, relating to developer performance and software quality, additional research in this area is encouraged to be undertaken.

The focus on the software development process characteristic of flexibility, particularly by agile development methodologies, has resulted in a concentration on certain aspects of software test. Crispin and Gregory (2009) referred to the emphasis on agile as being reflected in the associated software testing. Such testing is stated as being defined by the business experts' desired features and functionality, and not

generally by tests which critique the product. An explanation for this has been provided by Martin et al. (2007), in reference to the role of software test operating in an agile development, they stated that there is somewhat of a rejection of the latter phases of a traditional phased approach to software testing, which often tend to have a non-functional focus and are generally tests which do not easily conform to automation. This is explained as being a product of the test design process whereby the focus tends to be on tests relating to functionality and how different users would use the system. This view is backed up by Patel and Ramachandran (2008) who have argued that the general application of agile frameworks tends to attract a focus on functional requirements where there should also be a focus on other non-functional requirements such as *operability*, *observability*, *controllability*, *understanding*, *performance*, and *usability* of the software. Even though non-functional requirements are stated as playing a vital role in satisfying overall customer requirements, they are stated as not generally being covered by the exploration phase of agile based projects. A lack of focus on non-functional requirements, at the initial stages of the software development process, can prove increasingly difficult and costly to address at the latter stages of the process.

2.3.2 A Comparison of Traditional and Agile Software Development

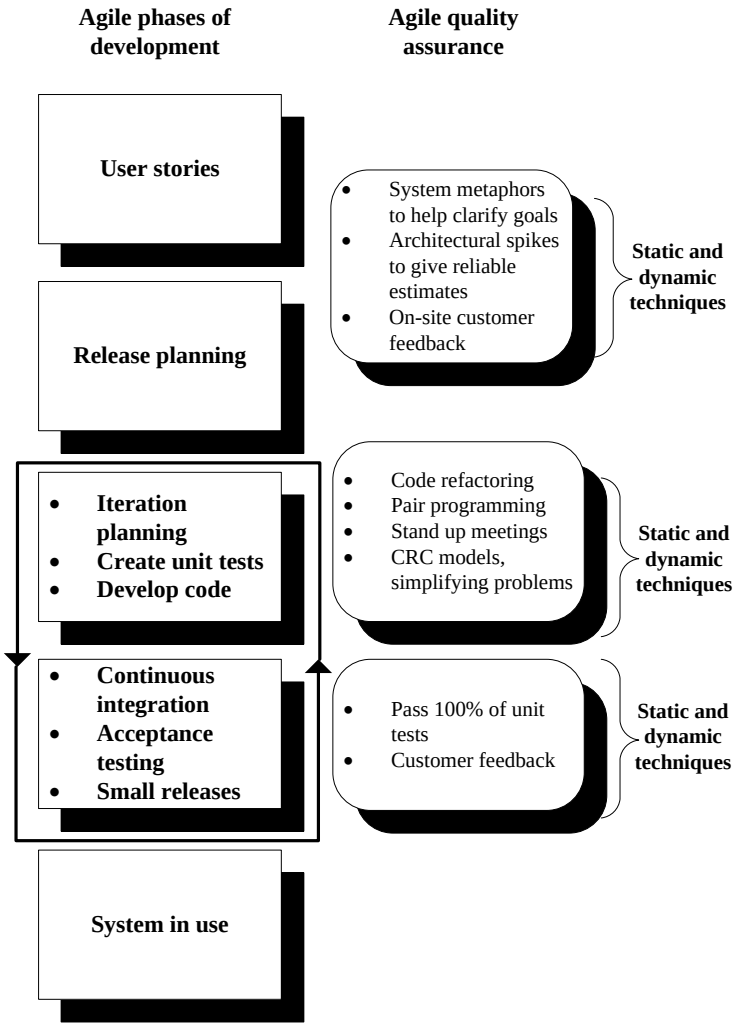
Huo et al. (2004) has provided us with a comparison of the waterfall development methodology and agile development methodologies in terms of quality assurance techniques.



(Huo et al. (2004)

Figure 2.2: Waterfall Approach from a Static/Dynamic Perspective.

Figures 2.2 and 2.3 present the general phases of development for both waterfall based projects and agile based projects, respectively, with the nature of the technique (static or dynamic) highlighted. This is interesting because this provides us with a perspective in terms of the core characteristics of the software development processes, from a quality assurance perspective (including test).



(Huo et al. (2004)

Figure 2.3: Agile Approach from a Static/Dynamic Perspective.

Notwithstanding the conflicting views on the appropriate development approach to follow, there are a number of fundamental activities that can be identified from section 2.3.1 of this chapter, which are common across traditional or agile development approaches. These are evident form the comparison as provided by Huo et al. (2004) (detailed in figure 2.2 and figure 2.3), and are associated with three principle activities:

1. *Software specification and design:* The functionality and constraints associated with the software must be defined. This may take the form of requirements definition and software and system designs or alternatively approaches such as user stories, system metaphors, architectural spikes, and release planning.

2. *Software implementation:* In line with the requirements, goals and designs, the software must be produced. This can be a planned, iterative, development process, or a planned, sequential, development process.
3. *Software verification and validation:* The software must be validated to ensure it acts in accordance with customer requirements or standards. Code verification can take the form of static checks such as code reviews, inspections, and peer programming, or dynamic approaches such as software unit testing and system testing. Validation can take the form of customer feedback and acceptance testing.

The *software implementation stage* is key in any software development environment but in the context of this research, of particular interest are the general software process activities of *software specification and design*, and *software verification and validation*. Software specification and design is important because, amongst other objectives, this activity facilitates the transfer of knowledge between two key stages of the software development processes, namely development and test. The importance of knowledge transfer to software development has been emphasised by Chau et al. (2003) and Cataldo and Ehrlich (2012). Chau et al. (2003) have referred to the importance of knowledge to all aspects of software development. It is stated that it is unlikely that all members of a software development team will possess all of the required knowledge for software activities such as requirments gathering, design, development, test, deployment, maintenance, and project coordination. Another area of importance which is discussed in greater details in a forthcoming chapter, relates to the importace of tacit knowledge which is associated with both traditional software development approaches, and agile development approaches such as XP (Boehm, 2002). It is stated that there is a risk of architectural mistakes because of an unrecognised shortfall in tacit knowledge, and that traditional or plan driven methods, reduce this risk by investing in life-cycle architectures and plans.

A downside of a formal approach to software development, are the costs associated with documentation updates, and the associated risks of such documentation being incorrect or not up to date. These views are also emphasised by Paetsch et al. (2003) who stated that the lack of documentation may present particular issues in the case of somebody leaving with key knowledge, and also suggested that tacit knowledge

transfer can become difficult in the case of complex projects (a view which has also been backed up by Turk et al. (2005), and Moe et al. (2012)). Another key difference identified by authors, between documentation associated with traditional development environments, and agile development environments, is the tendancy to focus on functional requirements in the case of agile documentation, and not necessarily devote resources to documenting requirements such as resources, maintainability, portability, safety or performance ((Paetsch, Eberlein, & Maurer, 2003), (Patel & Ramachandran, 2008)). The impact of both explicit documented knowledge and implicit knowledge on the software test aspect of the development process, which is key to this particular research, is discussed in greater detail in the next chapter.

Of primary interest for this research is the topic of *software verification and validation*. Huo et al. (2004) have stated that the encouragement of agile software development techniques to develop code early on in the development process, has invited many opportunities for quality assurace techniques to also be applied at an earlier basis. Dynamic activities such as the application of test driven development (TDD), and early acceptance testing by the customer, play a key role in maintaining software quality, along with static techniques such as code inspections, the development of user stories, the detailed consideration of architectural spikes, and the analysis of customer feedback, all of which are deemed vital to quality assurance. It is stated that, contrary to common perception, the frequency which quality assurance practices occur under agile methodologies, is greater than those proposed under the waterfall approach, but the key is in the application of those practices by development teams. The difficulty with making a comparison of the costs associated with the application of various development approaches, is something which is also referred to by Mitchell and Seaman (2009). They refer to the little empirical evidence which exists to provide an indication as to the cost, development duration timeframe, or quality benefits of one technique over the other. Tsui and Karam (2007) stated that testing is primarily carried out by three distinct groups:

1. *Software developers*: the role of software development testing is described as being primarily to create and run tests to verify that software programs run as intended and complete without major error.

2. *Software testers*: this role is described as involving technical persons whose role it is just to write and execute specific test cases with specific goals. It is stated that although development knowledge is extremely useful for testers, that it is a very different activity to that of software development, with completely different requirements. This view is also endorsed by Loveland et al. (2005). A major difference between the testing carried out by developers and testing carried out by professional testers, would appear to be that the role of software testers is often analyse test results and make assessments regarding software quality, often being called in to assist on making product release decisions.
3. *Customers or end-user testing*: it is stated that it is a good idea to involve users in testing in order to identify usability issues, and to expose the software to range of inputs in real world environments. User testing may also form the basis for software product acceptance decisions.

The focus for section 2.4, and subsequent sections, is on *software verification and validation*. Further discussions will take place from the perspective of software testing, involving independent software test teams, as distinct from testing driven primarily by development, or testing carried out by customers or end-users of the software. This is in keeping with authors views which are relating to traditional software development models, such as outlined by Royce (1970), Pfleeger (2001), Crispin and Gregory (2009), Talby et al. (2006) and Striebeck (2005), who specificaly have referred to the use and benefits of test professionals in an agile software development environment, particularly in the case of larger projects.

This acceptance of an independent test team working in an agile software development environment, is not necessarily in keeping with the views of all authors. Huo et al. (2004) and Patel and Ramachandran (2008), have outlined an agile environment, which makes extensive use of test driven development, proposing that developers are at the very least largely responsible for the development of software tests. The involvement of testers, aids the acquisition of requirements from customers, helping customers express their requirements as tests, as well as advocating quality on behalf of the cutomer, during the development process ((Pfleeger, 2001), (Crispin & Gregory, 2009)). Talby et al. (2006) have stated that independent testers allow a more

comprehensive test coverage, especially in the case of complex development projects. Such an approach also allows development to concentrate on developing code, as opposed to dedicating a significant portion of time on test case or test suite development.

By focussing on development environments which utilise independent test teams, there is an obvious dependence on knowledge transfer between development teams and test teams. Difficulties associated with geographical distributed development teams has been highlighted by Chau and Maurer (2004), and Lee et al. (2006). Lee et al. (2006) have highlighted the varying success with communication, in geographically dispersed development enviroments. It is stated that where there is less of an emphasis on explicit documentation, that geographical dispersion makes it increasingly difficult to share knowledge. The importance of knowledge sharing, and specifically tacit knowledge, in software development environments, has been emphasised by Ryan and O'Connor (2009). This is dealt with in more detail in the following chapter.

2.4 Software Verification and Validation

Verification and validation are described as important tools to enable a check to be carried out that a software product conforms to its requirements and specifications ((Tsui & Karam, 2007), (Sommerville I. , 2007), (Khan & Khan, 2014)). Sommerville (2007) stated that testing is the primary software validation and verification technique. Verification is described as confirming that system additions and modifications, made through the development phases, conform to system specifications, whereas validation is usually applied at the end of the project, to a complete software system and goes beyond checking that the system conforms to specifications, to validating that the software does as the customer expects it to do. Software testing, a dynamic approach to software verification and validation, is not a unique tool in this respect, in fact many static methods have also been shown to be beneficial in helping to ensure the quality of software e.g. software inspections, automated source code analysis, and formal verification (Delahaye, Kosmatov, & Signoles, 2013). However, these are performed against non-operational software, and cannot demonstrate whether the software is

operationally useful. Software testing is described as an important method for validating software usefulness, and checking software quality characteristics, such as functionality and reliability (Holzworth, Huth, & deVoil, 2011). Tsui and Karam (2007) have stated, that the level of required confidence, that all of the customers' expectations that will be met, is dependent on three main factors:

- *Software function*: How critical the software is to an organisation. An example is given that the level of confidence required for safety critical systems may be higher than otherwise necessary.
- *User expectations*: Prior to the 1990s, there was a generally low expectation of software and failure did not necessarily come as a surprise. However the author states that now more than ever, it is now considered unacceptable to deliver unreliable systems, so companies must therefore devote more time and effort to validation and verification.
- *Marketing environment*: When a system a system is marketed, the level of confidence required, will be dictated to a certain degree by the quality, price and supply of competing products.

To enable meeting customer expectations, the author refers to the complementary roles which software inspections and testing play in the software process, highlighting the fact that in you can only test a system when a program or executable is actually developed. Stated also is that requirements and design reviews are the main techniques used for error detection in the specifications and designs. Several methods are referred to which can be used for detection of errors in programs, both from a static point of view (verification and validation of non-running code e.g. via code reviews) and from a dynamic point of view (verification and validation of running code):

- *Testing* involves executing the software in a controlled environment and verifying that the output is correct.
- *Inspections and reviews*, which can be applied to programs or relevant documentation. These generally involve more than one individual in addition to the document or program creator. These are described as being labour intensive but an extremely effective method of finding errors.

- *Formal Methods* involve mathematical techniques used to prove that a program is correct.
- *Static analysis* involves analysing the static structure of a program or relevant documentation. Usually automated, this method can detect errors or error-prone conditions.

Such methods are common in both traditional and agile software development environments ((Huo, Verner, Zhiu, & Bahar, 2004)). As referred to in the introductory section, Sommerville (2007) emphasised that techniques such as software inspections, automated source code analysis, and formal verification, can only check the validity of a program is in accordance with the specifications, and cannot demonstrate whether the software is operationally useful (this view has been endorsed by Delahaye et al. (2013)). Software testing, a dynamic technique, is described as being the foremost method for software validation and verification, checking properties of the software such as performance and reliability. The fact that code coverage tools deal with static code and ignore operational context, is something which is seen as a disadvantage. Although code coverage tools provide developers with an excellent method of ensuring that tests execute against specific lines of code as planned, there are a number of problems which code coverage tools may not help address, such as bugs relating to running code, relating to specific timing events, and other events which occur as a result of code being executed in parallel (Loveland, Miller, Prewitt, & Shannon, 2005).

The importance of software testing has also been emphasised by other authors ((Wegener, Baresel, & Sthamer, 2001), (En-Nouaary, 1998), (Mattiello-Francisco, Martins, Cavalli, & Yano, 2011), (Yin & Ding, 2012)). Wegener et al. (2001), Mattiello-Francisco et al. (2011) and Yin and Ding (2012) have emphasised the merits of a structured approach to software testing, in terms of effectiveness and efficiency over an ad-hoc approach. A structured to testing has been provided by Eickelmann and Richardson (1996), who have highlighted key functions which software test environments have evolved to include over a period of time:

1. ***Test Execution*** includes the execution of the instrumented source code and recording of execution traces. The output of this stage includes test output results, test execution traces, and test status.

2. ***Test Development*** is essentially the development of a *test approach* which includes the specification and implementation of a test configuration. The output of this stage are the test suites including individual test cases, test input criteria, test documentation, and test adequacy criteria.
3. ***Test Failure Analysis*** includes behavior verification, and the documentation and analysis, of test execution pass/fail statistics. The output of this stage includes the pass/fail status and test failure reports.
4. ***Test Measurement*** includes test coverage measurement and analysis. Source code is described a typical instrument used to collect execution traces. Executed test runs have associated test coverage measures and test failure measures.
5. ***Test Management*** includes support for the complete test infrastructure, along with test environment state preservation. Test process automation usually requires a repository for the test infrastructure.
6. ***Test Planning*** includes the development of a plan relating to test case development. This provides the foundation for the development of test objectives. Detailed as part of test planning, are features of the system to be tested, risk assessment issues, organizational training needs, required and available resources, a comprehensive test strategy, outlining resource and staffing requirements, the roles and responsibilities, and the overall schedule. Development of a *test architecture* which outlines the required and available resources is also carried out at this stage.

Fundamentally, the model proposed by Desai and Shah (2011) relating to the functions of software test, is similar to that highlighted above, with the slight difference of an emphasis on a test environment preparation stage, as opposed to a test management stage. Notwithstanding that test management is an ongoing activity, which may be invoked at the start of projects also, the following order is proposed as the standard execution order of the aforementioned test related functions. This is also the order which they are discussed in the following section:

1. ***Test Planning***
2. ***Test Development***
3. ***Test Execution***

4. ***Test Failure Analysis***
5. ***Test Measurement***
6. ***Test Management***

In their classification of different types of testing, Walter and Grabowski (1999) specifically highlighted as important, the key aspects of *test objectives*, *test approach*, and *test architecture*. These aspects of software testing are used in the forthcoming sections to give an indication of the importance of structure to software testing. The consideration of *test objectives* is in reference to the consideration of expected behaviour of the system under stimuli. This can be categorised as functional or non-functional. Functional whereby there is correct system behaviour under stimuli which would be associated with operational circumstances and non-functional which could account for testing of general timing constraints, reliability, robustness, and possible organisational impacts such as ease of use, efficiency etc. Approaching the objective of non-functional testing could prove most difficult in the case of complex real time systems, because this arguably involves the correct behaviour of the system under failure, which can be due to an exhaustive list of reasons. The consideration of a *test approach* relates to the task of test case specification. Test cases and may be specified from a perspective of black box testing, white box testing, or a combination of both (discussed at the end of this chapter). The third consideration, as detailed by Walter and Grabowski (1999) refers to the *test architecture*. The authors describe this as being a combination of test equipment, all interconnectivity between elements of the system under test, and the actual system under test. Such architectures may also be of a distributed nature.

The aforementioned topics of *test objectives*, *test approaches*, and *test architecture*, are discussed in more detail in the forthcoming sections. Test objectives are discussed as part of test planning, test approach is discussed primarily as part of test development, and test architecture is discussed in the forthcoming section, as part of a discussion on test management.

2.4.1 Test Planning

Test planning has been described by Desai and Shah (2011) as involving the plan of the test case development, and the outlining of test objectives. As part of the consideration of test objectives, this section considers the many different phases of software testing, such as functional, regression, integration, product, unit, coverage, and user-oriented. All of the aforementioned are verification methods, which may be applied during or after the development phase ((Horgan & Mathur, 1996), (Huo, Verner, Zhiu, & Bahar, 2004)). As detailed at the start of this section, we are most concerned with testing carried out by independent test teams. Both of the aforementioned authors refer to phases of testing of having a focus on testing of the system functionality, testing of the software structure, or testing of the user view of the software respectively. According to Horgan et al. (1996), any of these methods may be applied to the various phases of software development.

The three general test areas identified by Berman and Cutler (2004) as encompassing any test process are *unit testing*, *integration testing*, and *system testing*. These three areas form the basis of the forthcoming discussions, in addition to a discussion on *acceptance testing*. This is in keeping with the high level view of testing from a perspective of testing of the software structure (unit testing), testing of the system functionality (integration of system testing), and testing of the user view of the software (alpha or acceptance testing). An insight into the reasoning for all of these different stages of test is provided by Loveland et al. (2005), who state that different test phases are designed to target different software bugs, and that no single phase is adept at catching all defects. Each phase is described as having its own limitations in terms of effectiveness, primarily due to defect visibility and often applied cost restrictions. The question is also posed as to "why not merge particular test phases?" The answer to this question is that, as described in the forthcoming section, although some of the test phases may appear quite similar, they actually carry out different, valid functions. Thus while the system test team carries out testing on the software, and a failure may block progress in the system test area, the goals of the test team covering functional testing are described as being that much different, that they can continue and may therefore not being prohibited from proceeding.

As referred to in the software development overview section (2.3), Tsui and Karam (2007) and Huo et al. (2004) have made reference to the three principle test groups as being *software developers*, *software testers*, and *end-users*. As well as addressing the different forms of testing from a developer, tester and user perspective, this section has also made reference to important software characteristic of reliability, also referred to by Walter and Grabowski (1999). The importance of reliability as a software system attribute has been emphasised by Cai (1998), who have stated that software reliability is the most important software attribute. The author ranks issues relating to software reliability alongside those of *cost*, *schedule* and *functionality*. The importance of reliability as a software characteristic has been emphasised by other authors also, such as Patel and Ramachandran (2008). In keeping with the common project goals of cost, quality and time to market, a discussion takes place at the end of this section regarding the limits associated with software test methods.

2.4.1.1 Unit, Stub, Module, or Function Testing

Software developers often create and run tests to verify that software programs run as intended and complete without major error. Yeates et al. (1994) described unit or program testing as a stage to ensure that all programs are fully functional. This is described in similar terms by other authors ((Horgan & Mathur, 1996), (Bentley & Whitten, 2007), (Tsui & Karam, 2007)). While there are agreements that unit testing relates to testing of modules, there is a slight difference between how authors categorise the unit test phase. Pfleeger (2001) stated that testing of individual component testing, often referred to as module, component or unit testing, verifies that the individual components operate as expected based on inputs. The purpose of this test phase is to verify code paths involving all inputs and outputs from logical code blocks such as functions, sub-routines, diagnostics etc. Tsui and Karam (2007) detailed a similar view. Bentley and Whitten (2007) made a distinction between the unit testing, and stub or module testing. Stub or module testing is what they refer to a test stage prior to unit testing, involving all sub-components associated with a program such as events or modules. They emphasise the importance of this stage stating that it is not beneficial to defer all testing until programs are completed. An important characteristic of the unit test stage is the requirement of test plans which are produced by developers, and generally verified by independent engineers.

Another level of testing relating to developers involves the testing of the interfaces between programs in the same functional area ((Horgan & Mathur, 1996), (Loveland, Miller, Prewitt, & Shannon, 2005), (Tsui & Karam, 2007)). This requires the testing of all interacting programs, ensuring that not only is data correct and happening in the correct sequence but also that specified response times are being adhered to. Bentley and Whitten (2007) have referred to this stage as integration testing, whereas Horgan and Mathur (1996), Pfleeger (2001) and Tsui and Karam (2007), have referred to this stage as also forming part of function testing. Loveland et al (2005) have made the distinction between product-wide integration of software modules (often described as system integration testing), and the integration of modules on a function by function basis, therefore they describe this stage as *Function Verification Test (FVT)* and not integration testing. This stage is described as being white box based, with testers focussing on testing functions, internal and external interfaces, operational limits, messages, crash codes and module and component level recovery. One particular benefit of this test phase, according to Loveland et al. (2005), is that it deals with modules collectively, focussing on the encompassing software functions, and often allowing testers to develop and execute detailed test scenarios which result in the execution of all aspects of the applicable code. The focus at this stage is whether the software performs as designed, and verification that it performs in line with customer expectations.

While there are a number of positives associated with this type of testing such as allowing a code coverage view whilst still being at a sufficient level to execute specific software functions, there are also some limitations. In this phase the test focus is generally from a basic functionality perspective, and thus testing may also be limited in terms of the stress which the system may be placed under, in comparison to the final deployed environment. The fact that this type of testing focuses on individual functions, and is therefore not verifying the interactivity and timing associated with the complete system, could be considered a limitation. There can be a considerable amount of work involved in testing all the functions of a system, but software test and automation tools can provide great assistance, in improving test efficiency and reducing costs. As distinct from unit, stub or module testing, function testing is often performed by a separate integration test team, providing an independent perspective

from that of a development team (Pfleeger, 2001). An important point has been raised by Tsui and Karam (2007), who stated that when testing a unit that depends on many other modules, that there may be a mix of unit and integration testing being carried out, thus there may well be situations whereby the software developers are carrying out some, if not all of the functional testing.

Another important point is raised applying to system testing, with the statement that when developing software components for use by other software components, on analysis, the system as a whole may constitute a traditional functional unit. This may result in the merging of function testing with system testing.

2.4.1.2 System or Integration Testing

Independent software testers are technical persons, whose role it is just to write and execute specific test cases, with specific goals. Although testers may be closely associated with development teams and may have a detailed knowledge of the software, the goals of the testers are not necessarily in line with that of development. Whereas the ultimate goal of development is to implement functionally correct software, the role of tester is to advocate quality on the customers' perspective, assisting development in achieving business value (Crispin & Gregory, 2009). It is stated that testers often analyse test results and make assessments regarding software quality, often being called in to assist on making product release decisions. System integration testing is described as a precursor to system testing which involves building the system from its components and testing the resultant system for problems that arise from component interactions (Sommerville I. , 2007). According to Sommerville, three different components are recognised as being involved in integration:

1. *Off the shelf components.*
2. *Reusable components that have been adapted for a particular system.*
3. *Newly developed components*.

Integration Testing

Integration testing then checks that integrated components are called correctly and that data is correctly transferred at the correct time across interfaces. It is stated that a *top down approach* could be taken whereby functional components are added in increments to an overall skeleton system. A *bottom up approach* to integration involves adding all infrastructure components such as network and database access initially with functional components being added subsequently. In both cases additional software is often necessary to simulate other components to allow the system to execute. An incremental approach to integration is advised, where possible, in order to enable easier diagnosis of errors. A recommended approach is to integrate the components that implement the most frequently used functions initially, thus ensuring that such components receive the most testing over the full development cycle. In reality however this may prove difficult, because features may be spread across multiple components, and thus all necessary components may have to be integrated to allow testing. Testing may reveal faults in interactions between components and repairs may involve changes to multiple components thus making the repair process more difficult. *Regression testing* is highlighted as an important part of integration, and involves rerunning tests relating to previous software increments, and running tests relating to new functionality. This is considered an easier process when development models such as XP are employed because of the upfront focus on test development.

System Testing

The Institute of Electrical and Electronics Engineers (IEEE) define system testing as testing a completely integrated system to ensure it meets its requirements (IEEE, 1990). Other authors define the system testing task as a set of activities intended to assess the performance and interoperability of the completed features of an application (or complete system) with respect to its requirements (and intended use) ((Miller, DeCarlo, Mathur, & Cangussu, 2006), (Bentley & Whitten, 2007)). The idea of a completed system is not necessarily always the case. In the case of an iterative development model being applied, system test may well be applied to a non-complete working system ((Loveland, Miller, Prewitt, & Shannon, 2005), (Sommerville I. , 2007), (Tsui & Karam, 2007). System test involves focussing on the software's function, but at a higher level than unit testing or integration testing ((Loveland,

Miller, Prewitt, & Shannon, 2005), (Bentley & Whitten, 2007), (Tsui & Karam, 2007)). Crispin and Gregory (2009) described testing such as system testing, as going beyond functional testing such as covered by test driven development or acceptance testing (dealt with in the next section), to dealing with other critical forms of testing such as load, performance, stress, and usability. Under this phase, system test views the software from the customer perspective, carrying out all activities such as all functional activity as well as configuration related activities such as upgrades, downgrades, installs.

System testing may also incorporate failure recovery from a variety of activities, to ensure that if failure does occur that the system handles such failure gracefully. The system test effort attempts to identify the most complex of system defects which may relate to a combination of certain events relating to specific timings. Heavy workloads and stress testing run over extended periods of time are described as increasing the risk of data integrity issues. Security defects and complex recovery defects are also targeted during this phase of testing. According to Loveland et al. (2005), system test has a goal of exposing architectural disconnects which may have occurred. This drives the system test stage to operate in an environment as close as possible to that of any potential customers. If virtualised environments are being utilised then they obviously have the benefit of cost reduction but any such environment should be capable of achieving its goals and objectives. A risk assessment should be carried out, regarding any deviation from customer deployed environments.

There may be difficulty associated with system testing when attempting to identify the source of defects using messages, logging, and other low level interfaces. Another difficulty may be the implementation of such a framework to cater for such activities. Once such a framework is in place to aid the identification of the source of any particular defects, then there are obvious positives to testing against a system which is similar to its proposed deployed state. A downside associated with system test environments can be the associated costs with building complicated hardware configurations in attempts to mirror the working environments of the most typical customers. Decisions have to be made in attempts to achieve system test goals, while meeting budget challenges. As previously referred to virtualisation is one area which should be explored in attempts to meet such challenges.

Regression Testing

Regression testing, described as a critical part of the system and integration phases, is described by Harman and Yoo (2007) as an activity performed to provide confidence that changes do not harm the existing behaviour of the software. Yeates et al. (1994) have provided a similar definition, but differentiate between ensuring the correctness of minor modifications which have taken place during system test, and the application of regression testing to maintenance phases to help ensure the correctness of modifications and enhancements which have taken place during such stages. Regression testing relates to the retesting of a modified software product, and as such has been considered a form of system testing (Yeates, Shields, & Helmy, 1994)), or may be considered as an independent phase of testing ((Horgan & Mathur, 1996), (Loveland, Miller, Prewitt, & Shannon, 2005)). Lin et al. (2012) have referred to the pressures associated with the regression test phase, stating that there can be considerable cost and time pressures associated with the regression phase of testing. Over the lifetime of a larger software product, the number of test cases could scale up quite considerably, as new versions of the software are released (the development methodology deployed has a considerable impact here, please see earlier sections for more detail on development methodologies). Running a complete test suite for every release can be both costly and inefficient, so software testers may be under pressure to construct a reduced test suite for regression testing, at a reasonable cost. The specific issue of test case optimisation is dealt with later in this chapter.

Performance Evaluation

As previously identified, Patel and Ramachandran (2008), have identified performance, along with reliability, as a key software quality indicator. Yeates et al. (1994) have made reference to system testing incorporating similar classes of testing such as performance driven testing, volume or stress testing (soak) testing. Yeates et al. (1994) have referred to the evaluation of performance, and state that any such evaluation requires relatively stable software, to allow for consistent results. As such, performance evaluation requires extensive test and debug, which has been carried out prior to execution. Loveland et al. (2005) have described performance testing as a method of evaluating performance, and state that this involves the validation of all response times or that the maximum transaction time period that can be met by the

system. This includes how long a system takes to respond to a user request, timing normal case paths through processing and exception cases. Performance testing is described as not only forming a necessary part of system test, but such testing may also apply to unit testing and functional verification testing. The main goal is described as being able to identify all system performance strengths and weaknesses, often compared against industry benchmarks. This type of testing may be related to how the software interacts with certain hardware or software bottlenecks. Virtualised test environments may often force the concentration on software bottlenecks. As is the case with system testing in general, performance testing can identify defects which require complicated solutions and thus may prove costly defects to resolve.

Load/Stress Testing

Yeates et al. (2004) described volume or soak testing as verification that the system can handle the specified maximum volume of usage, over a predefined period of time. Loveland et al (2005) have made a distinction between load/stress testing for performance verification and load/stress testing for the benefit of defect removal. Load/stress applied for performance analysis is to aid the identification of bottlenecks and to measure the execution speed of the software. The primary objective in this case is not to identify defects, but as previously stated, it actually depends on code stability for successful, repeatable, and consistent throughput, for specific events. A distinction is made between functional bottlenecks, unintended behaviour in software, which causes a reduction in expected performance and throughput, and performance degradation due to physical issues, such as memory or hard disk issues. Testing for defects through load/stress targets particular defects related to such things as complex combinations of events. To achieve this, the system test team applies load/stress to the software through a variety of workloads intended to mirror customer processing patterns. The distinction between this type of testing and performance based load/stress testing is that, as previously stated, whereas the performance team aims for clean, smooth, controlled test runs in order to gain precise, repeatable measurements, this type of testing uses load/stress as a testing tool for creating chaos. The aim is to recreate the most chaotic or complex of customer environments in an attempt to prove that the software is not stable. Even though similar tools may be utilised, they have opposing objectives.

Service Testing

Another possible aspect of regression testing is what is referred to as the service test phase. Service test is referred to as a primary approach to testing software fixes, both individually and bundled together (Loveland, Miller, Prewitt, & Shannon, 2005). It is described as not only affecting the fixes themselves, but also ensuring that those fixes don't have side effects that interfere with other areas of the software. It applies to unit testing, function verification testing, and system verification test levels. Typically fixes are validated by unit testing and/or function verification testing. The software load or bundle is then fed into the service test environment, which may or may not be similar to the product's system test environment. At this stage the service runs all the test scenarios and workloads, to ensure that no fix or fixes cause any software defects or performance issues. Service test is often limited by time constraints. There can be considerable pressure when fixes related to customer issues which are affecting customer business, are going through the service test phase. Service testing can be considered an efficient method of carrying out service testing on released software. This involves the grouping of software fixes into periodic releases rather than having extensive service testing being carried out on many separate releases.

This section has dealt with numerous forms of testing which are conducted by testers. Testing from the perspective of eventual customers or end-users of the software (referred to as *acceptance testing* or *alpha testing* in this particular research) was not discussed in this section, but has the obvious benefit of the system being tested by the natural end-user, in ideally a similar environment to that of a finally deployed system, Tsui and Karam (2007).

2.4.1.3 Acceptance or Alpha Testing

The importance of testing from the perspective of the end-user has been emphasised by many authors ((Royce, 1970), (Tsui & Karam, 2007), Ko et al. (2011)). This type of testing often forms the basis for software product acceptance decisions (acceptance testing), and described as a key stage of testing for agile development approaches ((Martin R. , 2003), (Huo, Verner, Zhiu, & Bahar, 2004), (Crispin & Gregory, 2009)). Martin (2003) has stated that acceptance tests verifies that the system as a whole works and that the customer requirements are being met. Tsui and Karam (2007) stated that it

is a good idea to involve users in testing in order to identify usability issues and to expose the software to range of inputs in real world environments. If the users are from within the developing organisation then this is referred to as "Alpha Testing", whereas if the users are from outside of the developing organisation, then this is referred to as "Beta Testing". The role of alpha and beta testing has been detailed by Loveland et al. (2005) and Tsui and Karam (2007), although Loveland et al. (2005) does describe alpha testing as a phase of integration testing but the goal is similar to as described in this section. Beta test broadens the exposure of the software to a range of customer environments, giving access to each customer's perspective on the software's impact to its business during and after deployment. This serves as an important phase in preparation for General Availability (GA) of the software, whereby the software is fully released to customers. The downside associated with Beta testing, is that it may be difficult to cover every possible environment and the amount of time a Beta release may be active prior to the software going GA may be limited.

A clear distinction is made between the role of system testing and the role of alpha testing. While system testing ensures that new software doesn't introduce major incompatibilities with prior test levels, the role of alpha testing is to assess whether it is possible to migrate to a new version of software, without disrupting the flow of work in a simulated customer environment. Therefore *alpha testing* is dependent on earlier test phases extracting lower level bugs, and all significant stability problems. Loveland et al. (2005) stated that if the alpha test team spends their time predominantly finding system specific functional issues then there is a risk that interoperability issues may not receive adequate investigation. Another important point made in relation to integration test, is that while the alpha team attempts to achieve a customer-like environment, it can't necessarily be all-inclusive but should be representative. The integration team's effectiveness is limited by the quantity and quality of customer information at its disposal, to aid the testers understanding of customers work environments, and how they choose their software packages to solve their business problems.

There are similarities between this stage of testing and general system test, because of the goal of identifying defects relating to timing, serialization, recovery, and integrity, but the authors argue that the bugs primarily surface due to the new context, which is

only one component of a bigger solution. One effect of striving to emulate customers work environments is that there may also be a necessity for professionals in specific areas such as systems administration, database administrations, application development and deployment etc. Using this method, employees have a greater chance of encountering the issues that may arise at a customer site. Such a process can also lead to defining best practices and product deployment documents, something which can add help customers in their deployment and usage of the system.

This section discussed the test process from a test planning perspective, outlining the potential objectives which would be considered in the development of a test plan. After defining a test plan and the associated test objectives, the next stage, as outlined by Desai and Shah (2011), is the consideration of test development. This is stated as involving the development of a test approach and test suites. These are developed in line with the previously defined objectives.

2.4.2 Development of Test Suites and Test Cases

Test development is described as involving the specification and implementation of a test configuration, which results in test suites, and any associated documentation ((Eickelmann & Richardson, 1996)). In line with the views of Walter and Grabowski (1999), *test approach* is discussed here as an important aspect of test development. It is described as a key element in any software testing strategy, and as primarily being concerned with the method of test case specification. As stated in the introductory section of this chapter, Horgan et al. (1996) have referred specifically to the testing methods of functional, coverage, and user-oriented, and link these phases directly to the testing of the system functionality, testing of the software structure, or testing of the user view of the software, respectively. An approach to any test method may be from a perspective of black box testing, white box testing, or a combination of both (Walter & Grabowski, 1999). What is generally referred to as black box testing, is where tests are specified with limited knowledge of the internal workings of the system, and test cases are generally derived from related specifications, such as functional specifications, system or feature designs etc. White box testing involves testing of the structure of the software via test cases, which involves required

knowledge of the program code, with test sequences being derived on analysis of the software structure. Littlewood et al. (2002) have also referred to the relationship between white box testing and black box testing. Walter and Grabowski (1999) refer to a hybrid form of black box testing and white box testing, which they refer to as grey box testing. Grey box testing is referred to as utilising the specifications for test case development, but with analysis of the software structure also taking into account during test case development. White box or coverage testing uses the structure of the software to measure the quality of testing. The authors describe this white box testing as being particularly important in the estimation of requirements such as reliability. The aforementioned authors go on to describe white box testing methods as including:

- *Statement Coverage*: Statement coverage involves the design of test cases so that each statement or block of code is planned to be executed at least once.
- *Decision Coverage*: The principle of decision coverage is that each decision in each program is covered at least once.
- *Data Flow Coverage*: Data flow coverage directs the tester to construct test cases which cover both the data definition and the subsequent value usage.
- *Mutation Coverage*: Mutation testing is described as involving the testing of all non-equivalent mutations of any program P. A mutant is described as being the product of a change to P, in accordance with a given set of rules.

A distinct advantage of these methods is that each of them provides adequacy criteria, against which a test can be evaluated. Test data which is data coverage adequate is also said to be decision adequate. Similarly, test data which is stated as being mutation adequate, is also said to be data adequate. Functional testing does not provide any such precise and measurable criteria, according to the authors. In fact the authors state that even after extensive functional testing, that test data cannot be shown to data flow adequate and therefore cannot be shown to be mutation adequate. It is stated however that for several types of errors, that structural testing is not sufficient but functional testing is. Furthermore, functional testing is described as the first step in verifying that the specific functions of a program perform correctly.

Mattiello-Francisco et al. (2011) and Yoo and Harman (2010), have highlighted two main aspects of the any approach to software testing:

- *The role of operational profiles*
- *Test case selection problems*

A key concern, which the aforementioned authors refer to, is that the authors maintain that traditional methods for automatic test generation are based on exhaustive black box testing, and as a direct result face test case explosion when dealing with complex communicating subsystems. This is also backed up other authors ((Zheng, Alager, & Ormandjieva, 2008), (Lin, Chou, Lai, Huang, & Chung, 2012)). In keeping with the issues identified in the introductory section, Lin et al. (2012) have made reference to the cost and time-to-market pressures associated with repetitive software testing. A solution which is proposed by Mattiello-Francisco (2011) is the development of an operational profile to guide software testing, by progressively breaking down system usage. Occurrence probabilities of the system operations can be based on their operational usage, allowing proportionally more time to be committed to those operations whose occurrence probabilities are higher. Walter & Grabowski (1999) have also referred to the lack of practicality regarding validation of responses for all input/output combinations of systems, stating that the number of state/input pairs is generally infinite. Management of test cases through various approaches such as *test case prioritisation*, *test case selection*, and *test case minimisation*, are other common methods for dealing with test case explosion, (Yoo & Harman, 2010). These approaches are in response to the impracticalities associated with providing complete test coverage for software systems, referred to in the introductory section as being highlighted by Myers (1979). As referenced in the previous section, there is also a discussion in this section on the importance of reliability to any software testing approach, from a perspective of both black box, and white box testing.

2.4.2.1 Operational Profiles

Mattiello-Francisco et al. (2011) have referred to the use of operational profiles as which attempt to model the intended usage of the system, in terms of operations and occurrence probabilities. The use of operational profiles is also referred to by other authors ((Horgan & Mathur, 1996), (Desmoulin & Viho, 2007), (Sommerville I. , 2007)). An operational profile approach to system testing involves the specification of

the intended usage of the system, often dealing with the system from a functional requirements level, in order to break down the intended usage. It is stated that a test model based on operational profiles, defines test effort of system operations, in relation to their operational use, with proportionally more effort being applied to those operations which have a higher occurrence probability. An operational profile is also seen as key to reliability estimation, reflecting how the software will be used in practice, enabling the specification of classes of input and the probability of their occurrence ((Horgan & Mathur, 1996), (Littlewood, Popov, & Strigini, 2002), (Cai K. , 1998)). Both Horgan and Mathur (1996) and Mattiello-Francisco (2011) detailed similar steps in the development of an operational profile. A customer profile is developed first based on input from perspective customers. This profile is refined in a number of steps to develop an operational profile. Test cases are selected in line with a particular operational profile based on occurrence probabilities. One test framework proposal based on service prioritisation involves the following steps. Firstly, detailing of a service profile relating to deployed usage. Mattiello-Francisco et al. (2011) have highlighted the relative compensation through test effectiveness based on effort being applied at the prior stage of operational profile development. The authors found, that there is a positive relationship between significant effort being applied when detailing service profiles relating to deployed usage, and compensation in the effective use of the test purposes, thus leading to more effective testing. A solution to the aforementioned issue of exhaustive list of possible test combinations, is also put forward by the aforementioned authors, who suggest a proposal of selection of major and minor timing deviations, thereby enabling them to emulate a situation of early or late messages, in addition to covering test purposes relating to lost, rushed, or duplicated messages. Despite the benefits, there are a number of difficulties highlighted by Horgan and Mathur (1996) associated with the employment of operational profiles:

1. **Inadequate test set** – Black box testing based on an operational profile, inevitably means that tests have been based on the features of the profile. An issue arises when a profile has not properly detailed all features, or when feature usage has been incorrectly estimated. The problem with such a strategy is that the adequacy of such a test set relies on the accuracy of the data relating to statistical sampling, used to develop the operational profile. This approach

does not account for the fact that an inaccurate profile may result in a poor test set. This point is echoed by Loveland (2005) who stated that if the system is to be fault tolerant, then the probability of failure of application modules needs to be determined. This can be quite difficult to achieve with new software, or indeed with new features. Such failure probabilities may depend on well understood phenomena, or not so well understood phenomena. Lack of customer base knowledge is likely to add a certain degree of uncertainty to the occurrence of probability estimates of features. On a similar note Sommerville (2007) cited difficulties associated with developing operational profiles when software is new and innovative, but also refers to the problem of operational profiles changing as the system is used, stating that as users become more confident with a system, they often use it in more sophisticated ways. Due to this reason it is difficult to be confident about the accuracy of an operational profile.

2. **Coarse features** – Although black box tests may have been constructed to exercise a feature thoroughly, there is often no measure of how well the feature has actually been exercised. There may in fact be areas of the code which has not been exercised, even though the feature occurs with a high probability in the operational profile. This is more likely to happen with random selection of test cases from the input domain of the tester is generating test cases manually, without knowledge of how well the code corresponding to this feature has been exercised to date. Horgan and Mathur have made reference to empirical data relating to two particular applications which had been tested extensively over several years. This data indicated that tests generated manually, using knowledge of program features and the functions used to implement them, is sufficient to obtain a high level of code coverage. On the other hand inadequate testing is likely to result in misleading failure data, and inaccurate reliability estimates, even assuming an accurate operational profile.

3. **Interacting features** – In a larger system, features tend to interact in a variety of ways. A simple form of interaction is when for instance, feature f1 works correctly when exercised before exercising feature f2, but not otherwise. The greater the number of features, the more complex and difficult it becomes to

check systematically the interaction of these features. Failure to check for faulty interactions may generate misleading failure data, leading to inaccurate reliability estimates. A similar point has been made by Loveland (2005) regarding feature interactions and the resultant complexity from an operational profile perspective. The tester may have no idea regarding feature granularity and the amount of lines of code involved per feature.

Operational profiles are described as one tool which can be used to increase the efficiency and effectiveness of software testing. The next section deals specifically with the task of software testing and difficulties associated with *test case selection* problems, focussing on *test case prioritisation*, *test case selection* and *test case minimisation*.

2.4.2.2 Test Case Selection Problems

The management of large numbers of test cases is something which numerous authors have referred to ((Zheng, Alager, & Ormandjieva, 2008), (Yoo & Harman, 2010), and (Lin, Chou, Lai, Huang, & Chung, 2012)). Running a complete test suite for every release can be both costly and inefficient, so software testers may be under pressure to construct a reduced test suite for regression testing, at a reasonable cost. The underlying assumption of running a reduced test suite while maintain quality goals, according to Walter and Grabowski (1999), is that if a system operates correctly for selected test cases, then it will operate correctly for all possible state/input pairs. Yoo and Harman (2010) have referred to the difficulties relating to test suite prioritisation, test suite selection, and test suite minimisation. Test suite prioritisation is described as driven by a desire to order test cases, enabling early maximisation of some desirable properties, such as the rate of fault detection. Such an approach ensures that the tester obtains maximum benefit, even if the testing is prematurely halted at some arbitrary point. The approach is first credited as being mentioned by Wong et al. (1998). Harold and Rothermal (1999) are credited with proposing and evaluating the approach in a more general context. To overcome the difficulty of not knowing fault detection information until testing is finished, test case prioritisation techniques instead hope that early maximisation of a certain chosen surrogate property will result in the maximisation of earlier fault detection. It is stated that in the case of a controlled

regression testing environment, the result of prioritisation can be evaluated by executing test cases in accordance with the detection rate. Lin et al. (2012) have provided six algorithms which are implemented by a database-driven method to reduce the size of test suites, with experiments being conducted by an automated production system which provides information on code coverage traces and execution times for each test case.

According to Yoo and Harman (2010), the test selection approach is essentially similar to the test suite minimisation approach; both problems are about choosing a subset of test cases from the test suite. The key difference is described as being whether the focus is on changes to the system under test. Test suite minimisation is often based on metrics such as coverage measured from a single version of the program under test. By contrast, in regression test selection, tests are often selected because their execution is relevant to the changes between the previous and the current version of the system under test. Therefore the approaches to test case selection are *modification-aware* with regards to emphasising the coverage of code changes. Rothermal and Harrold (1994) are credited with introducing the concept of modification-revealing test cases, between the original and the new release of a program. Rothermal is also credited with adopting a weaker criterion that selects all the *modification-traversing* test cases. A test case is modification-traversing, if and only if, it executes new of modified code in the new release of a program, or attempts to execute formerly existing code, removed from the current software. Lin et al (2012) have stated that this approach led to a premise that selecting a subset of modification-traversing test cases and the removal of test cases that are guaranteed not to reveal faults in a new release of a program is possible. Thus an approach to the problem of regression test selection was introduced by Rothermal and Harrold (1997). Though still not safe for detecting all possible faults, this approach provides a method of selecting modification-traversing test cases into a reduced test suite.

Although the above section refers to the main consideration of code coverage when carrying out a test case minimisation assessment, Lin et al. (2012) have stated that the criteria for selection of test cases may include:

- *Coverage criteria.*
- *Resource constraints.*

- *Fault detection capability.*

Lin et al. (2012) have stated that many regression test selection algorithms are based on code coverage or fault density capabilities. It is pointed out however that many of these algorithms demand a long execution time, with huge numbers of test cases often existing, when dealing with a large body of code. The potential for large volumes of test cases are something which is highlighted by other authors also ((Zheng, Alager, & Ormandjieva, 2008)). Through concentration on a function level of granularity, there are two metrics which are identified as important:

- *Test Intensity* - The percentage of test cases covered by a function.
- *Function Reachability* - The percentage of functions reached by a test case.

Lin et al. focus on providing a solution to the problem of how to select test cases, as part of a reduced test suite, yet still retain tests to effectively reveal faults. As part of a survey carried out by Yoo and Harman (2010), three test optimisation problems are highlighted. Two of these problems have already been referred to, namely test suite prioritisation and test case selection. A related third issue relating test suite minimisation is also referred to in this section. Horgan and Mathur (1996) have made reference to some considerations to be made when selecting tests. They state that a test case is defined as being useful, only if it increases some type of coverage. This has the potential to carry out execution of software, relating to what is referred to as disjoint subsets or partitions, described as causing a particular program to behave different under identical test conditions. There is a reliance on different test methods to expose such partitions. Without consideration of the code being covered during test execution, it is stated as being difficult to determine the usefulness of a test. Another point raised, relates to the consideration of rare events. For any given test case, a failure is considered a rare event, if the probability of occurrence is arbitrarily small.

Coverage based estimated, have been found to be more realistic to the ones that ignore coverage data. This is expected to lead to an increase in testing effort to raise the estimated reliability to a sufficient satisfactory level. Secondly a study of coverage helps the tester construct new test cases, in addition to the ones constructed during functional testing. Such test cases are likely to reveal faults that remained uncovered

during functional testing, based perhaps on the operational profile. Thus, failures that may have proved rare events during operation may in fact occur during testing.

The test suite minimisation problem seeks to identify redundant test cases and to remove them in order to reduce the size of the test suite. Lin et al. (2012) have stated that this method is also referred to as "test suite reduction", inferring that the reduction is permanent. In an effort to counter the negative views which may exist regarding test case reduction, an empirical study was conducted by Wong et al. (1998), to determine the relative importance of the size and coverage attributes, in affecting the fault detection effectiveness of a randomly selected test set. Results from the study conducted by Wong et al. indicate that as the size of a test set is reduced, if the code coverage is kept constant, then there is little or no reduction in the fault detection effectiveness of the reduced test set. Yoo and Harman have referred to a minimal hitting set algorithm (Harrold, Gupta, & Soffa, 1993), which categorised test cases according to the degree of essentialness. The hitting set algorithm is based on the assumption that each requirement can be satisfied by a single test case, which according to the Yoo and Harman (2010), may not be true. An example is given of a test requirement that is functional, rather than structural, and requires more than one test case to be satisfied. This means that the minimal hitting set formula no longer applies, and the functional granularity of test case needs to be adjusted accordingly, which may involve either:

1. *A view involving a higher level of abstraction being taken*: such an approach results in each test case requirement being met with a single test scenario composed of relevant test cases.
2. *Division of larger functional requirements*: under this approach functional requirements which demand multiple test cases, will be divided into smaller sub-requirements which can be serviced by individual test cases.

This problem is described as being NP-complete in that there is no known efficient method of locating a solution. Thus Yoo and Harman encourage the application of heuristics i.e. a solution that is accepted which achieves an acceptable, but is not necessarily the optimal solution. Chen and Lau (1998, 1998b) applied GR and GRE heuristic algorithms, which are developed depending on the essential, the 1-to-1

redundant, and the greedy strategies (G: greedy strategy, E: essential strategy and R: 1-to-1 redundant strategy). The aforementioned authors are described as defining essential test cases as the opposite of redundant test cases. If a test requirement r_i can be satisfied by one and only one test case, then the test case is an essential test case. On the other hand, if the test case satisfies only a subset of the test requirements satisfied by another test case, it is considered a redundant test case. Based on this Yoo and Harman summarise the concepts of GR and GRE as:

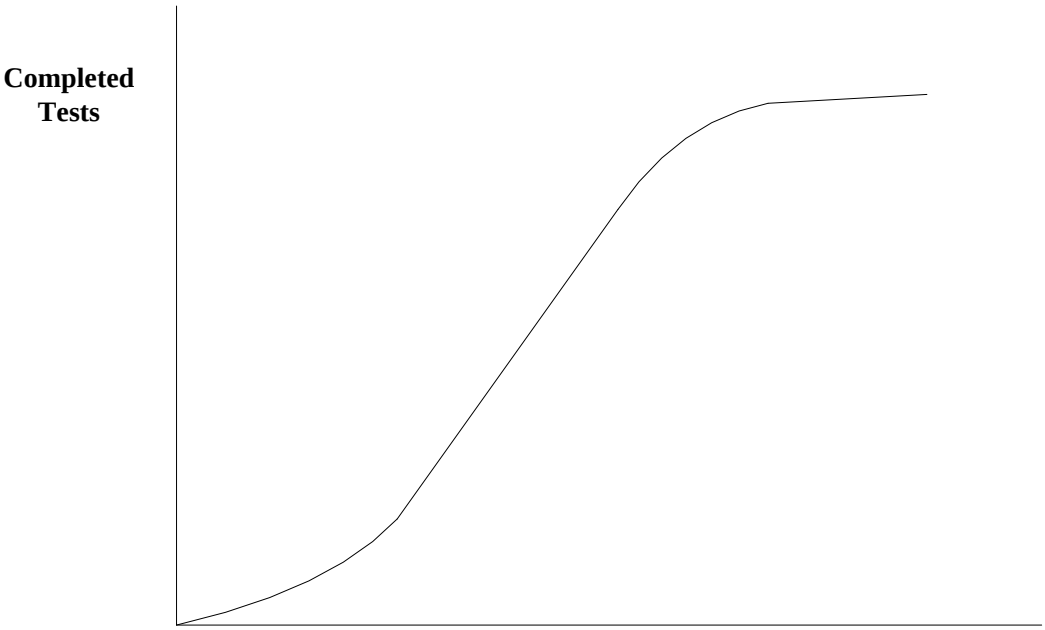
GE heuristic: first select all essential test cases in the test suite; for the remaining test requirements, use the additional greedy algorithm, i.e. select the test case that satisfies the maximum number of unsatisfied test requirements.
GRE heuristic: first remove all redundant test cases in the test suite, which may make some test cases essential; then perform the GE heuristic on the reduced test suite.

It is suggested that no single technique is better than the other. This is a natural finding, because the techniques concerned are heuristics, rather than precise algorithms. Wong et al. (1998) have adopted a heuristic approach to regression test suite minimisation, and conclude that there are at least two attributes that determine the fault detection of a given test set. The first attribute identified, is the size of the test set (measured as the number of test cases). Code coverage is also identified and is measured by executing the software across all elements of test set. The fault detection effectiveness of the test set is the ratio of the number of faults guaranteed to result in software failure, when executed on the test set, to the total number of faults present in the software. At the start of this section, the importance of operational profiles was mentioned. The strong link between operational profiles and reliability was also referred to as being highlighted by Horgan and Mathur (1996). The development of a test environment and test architecture is discussed in more detail in the following section.

2.4.3 Execution of Test Cases

Test execution has been described as being an obvious necessity for any test process, facilitating software debugging, and important activities such as reliability estimation

to be carried out (Eickelmann & Richardson, 1996). Test execution may be impeded by certain defects in the code and another difficulty highlighted is the saturation effect (Desai & Shah, 2011). The saturation effect is something which is referred to by Horgan and Mathur (1996) as affecting all testing methods. An understanding of this effect is described as a prerequisite to realising the shortcomings of any test model. The saturation effect relates to the tendency of an individual testing method to attain a limit in its ability to reveal faults in a given program. It is this limit which may cause over or underestimates of reliability, using existing models. As a test phase progresses, test information becomes increasingly available regarding necessary resources, failure data etc. Loveland et al. (2005) have referred to the saturation effect in relation to an iterative approach to testing, stating that there's always one big question: "how do you know when you are done?" The authors describe the measure of progress for traditional software testing, consisting of a non-iterative cycle between development and test. They describe this as the classic pattern following an "S" curve (figure 2.4). Progress is initially slow but the number of tests completed rises quite rapidly. Towards the end of the test phase, successful completion dwindles as testing awaits final fixes and tests such as performance and reliability tests are nearing completion.



(Loveland (2005))

Figure 2.4: Test Completion Progress.

Horgan and Mathur (1996) discussed the impact of the saturation effect on the complete software test process. They state that a program contains a certain number of faults. As testing proceeds the number of remaining faults decreases. However when applied, each testing method has a limit on the number of faults which it can reveal for a given program. Figure 2.5 is provided by Horgan and Mathur to give an indication of the saturation effect, the test effort associated with a particular test method and the faults revealed.

Faults versus Testing Effort (Horgan and Mathur (1996)).

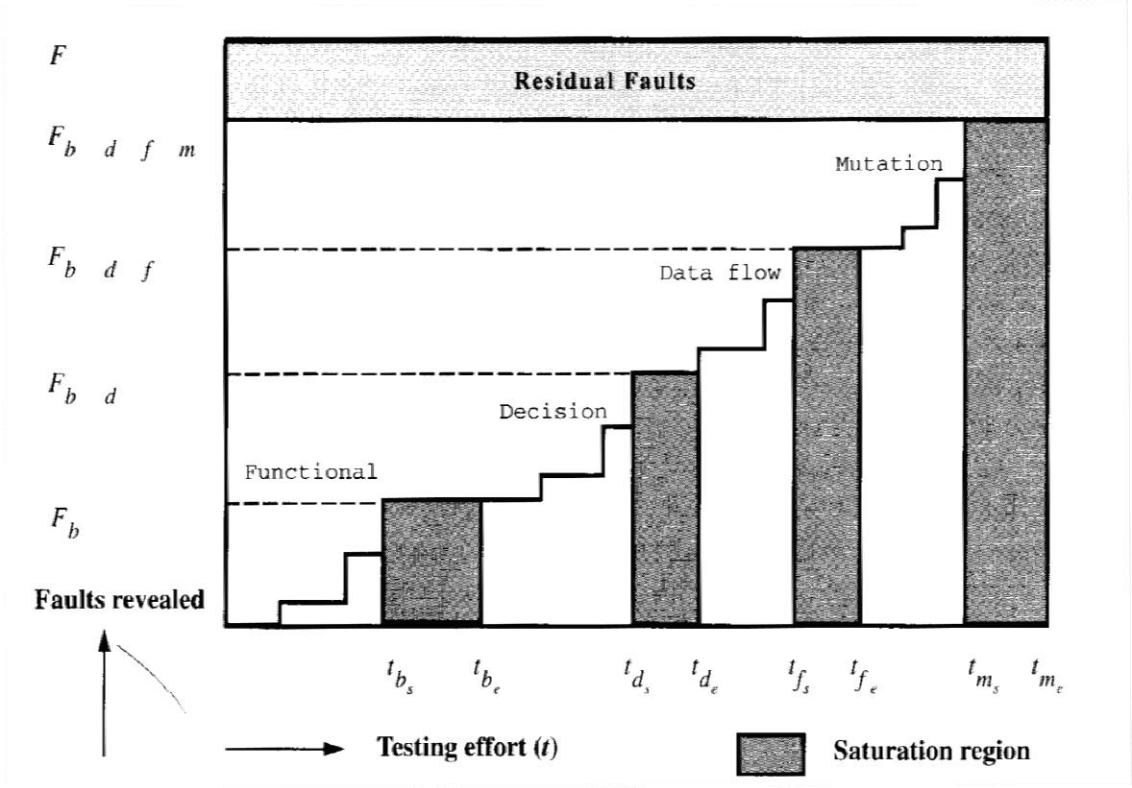


Figure 2.5: Faults versus Test Effort.

F_x relates to the number of faults revealed, t_x relates to the test effort associated with a particular test method $_x$ with an associated start $_s$ and end $_e$. The particular test methods have previously been discussed in this section under test planning. The authors maintain that each testing method has a limit on the number of faults that it can reveal for a given program. For instance in the case of functional testing, this limit is assumed

to have been reached after t_{bs} effort has been expended. Also functional testing has revealed F_b out of F faults when its limit has been reached. The authors state that in practice a variety of criteria, both formal, such as *reliability estimates,* and informal, such as *market pressures* are applied to terminate testing. Once the limit has been reached, if no additional faults are found and that a tester, testing continues testing without the discovery of any more faults to t_{be}. The reliability estimate can be improved by increasing the number of test cases executed in the saturation region. Switching between test methods is assumed to occur at t_{xe}, where x refers to the particular test method. Using the above aforementioned example in figure 2.5, after testing has completed, there are a total of $F_{bUdUfUm}$ faults revealed. There is a general assumption with the model provided by Horgan and Mathur, that each test step will reveal an increasing number of faults i.e. $0 \leq F_b \leq F_{bUd} \leq F_{bUdUf} \leq F_{bUdUfUm} \leq F$.

The previously mentioned assumption is backed up by analysis of test data which enabled the conclusion that intensive functional testing may fail to test a significant part of the code, and therefore may fail to reveal faults in the untested parts of the system. The authors use this observation to justify the claim that the saturation effect is exhibited by functional testing, and that coverage data must be used during reliability estimation (figure 2.6). Another consequence of the saturation effect according to Horgan and Mathur, is that it can lead to an overestimation of reliability. This may occur if for example the Musa model was being utilised whereby increasing inter-failure times usually results in an increase in an estimate of reliability, $\bar{R}$.

Test Saturation Points (Horgan and Marthur (1996))

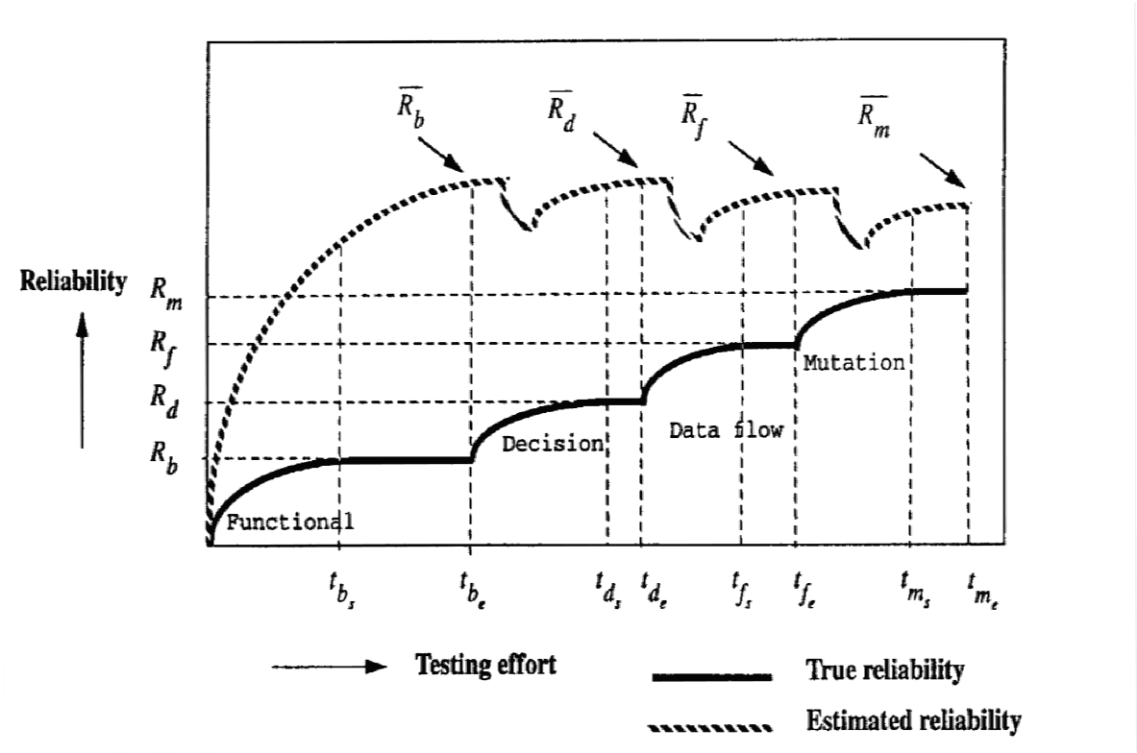


Figure 2.6: Test Saturation Points.

An assumption is made that the reliability estimate is a stochastically increasing estimate, implying that even though it may fluctuate, that it will eventually increase if the number of remaining faults decreases. Figure 2.6 indicates that as faults are discovered in the various test phases, the estimated reliability, $\overline{R}$, increases. As the testing progresses throughout a particular phase, faults are discovered and the value of $\bar{R}_x$ increases. In general it is not possible to detect the saturation point and thus testing may continue well past this point, increasing $\bar{R}_x$ but not necessarily R_x. This is explained by the continuation of testing with no new faults being detected and can lead to a considerable overestimation in reliability. This effect can occur when other test methods such as white box testing are applied also. Thus over a number of subsequent test phases considerable overestimation in reliability may occur.

Fault Removal Points (Horgan and Marthur (1996))

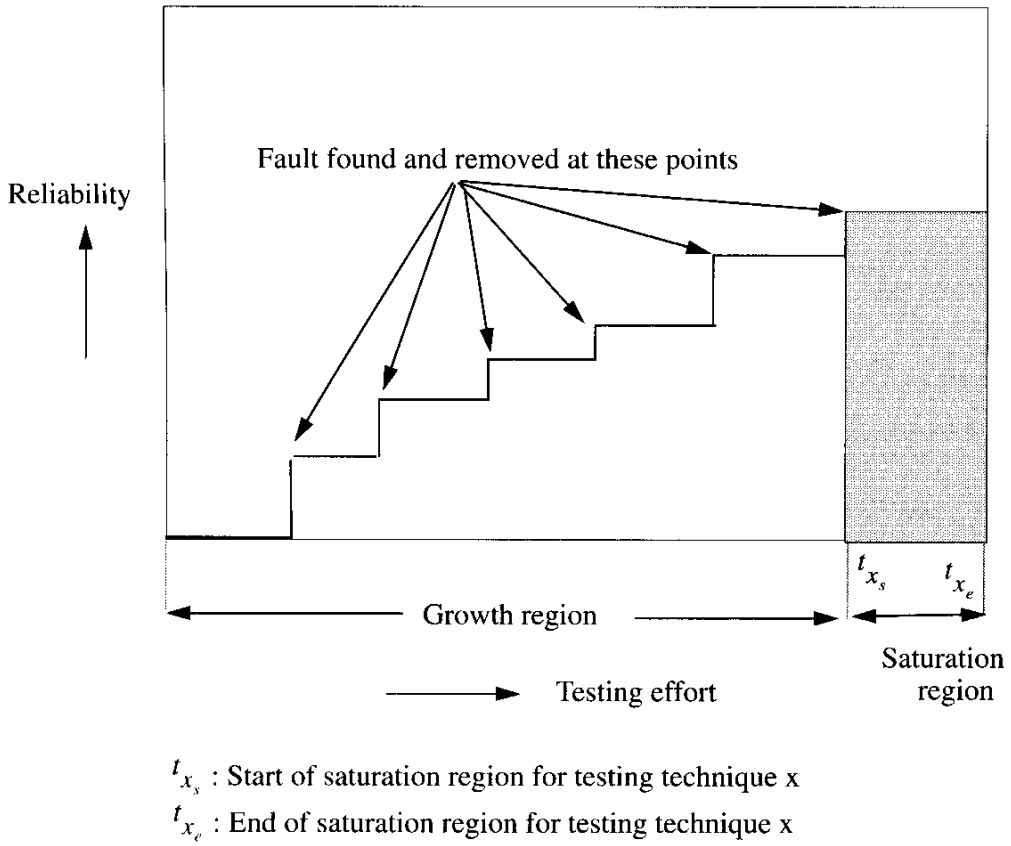


Figure 2.7: Fault Removal Points.

The example in figure 2.7 is of course based on faults being found and fixed on a gradual basis, whereas in fact the reality is more likely to be as detailed in figure 2.5, based on testing effort relating to CPU time and being carried out on a phase basis. This stepwise rise of reliability causes the considerable fluctuation in reliability estimation, $\bar{R}$.

Another difficulty associated with identification of the saturation effect, is highlighted by Loveland et al. (2005). When utilising some test methodologies such as Algorithm Verification Test (AVT), no new test phase can be considered complete, until all or nearly all of the tests are successful. Thus the plot of tests test progress is quite slow until finally a significant amount of progress is achieved regarding tests completed (figure 2.8). The authors state that methods such as this are particularly difficult to

recognise the point of significantly diminishing returns from testing or saturation point.

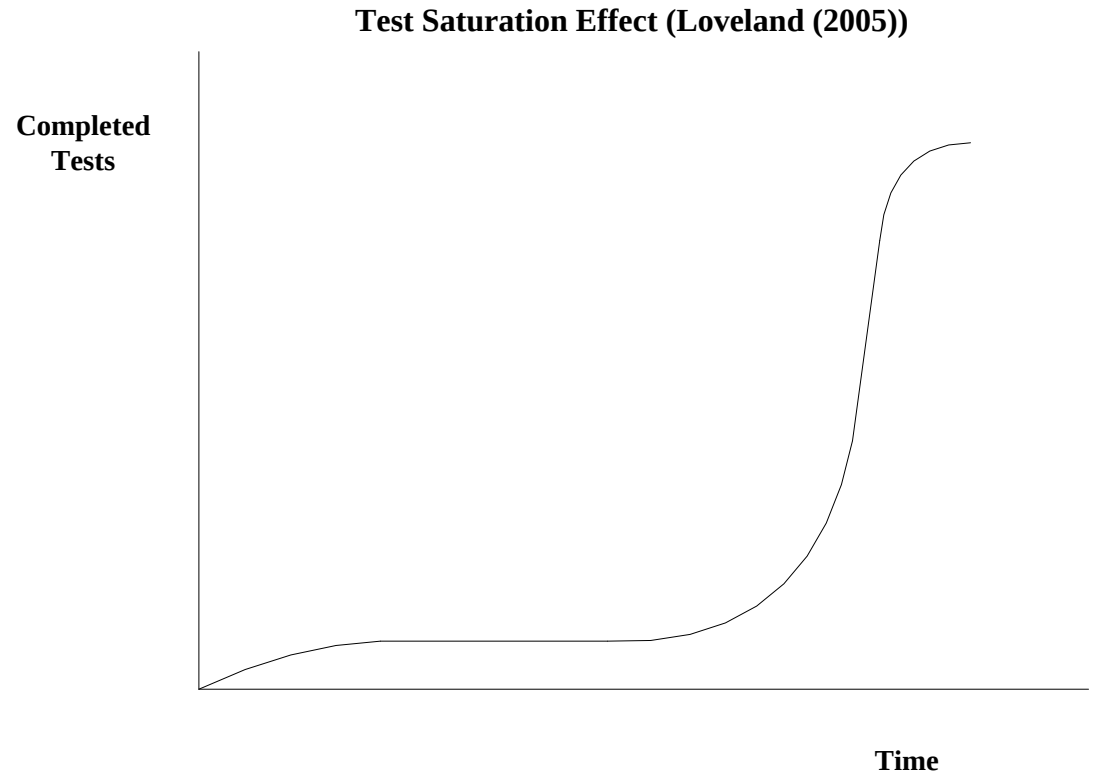


Figure 2.8: Test Saturation Effect.

Loveland suggests charting progress against defined goals, breaking the definition of the test into logical chunks. For each chunk you can test whether the code is available, the test is underway, and that a particular algorithm has met a predefined exit criteria. Closely integrated with test execution is test failure analysis which can be used to determine overall software quality.

2.4.4 Failure Analysis of Test Results

Eickelmann and Richardson (1996) have referred to test failure analysis as relating to the verification, documentation, and analysis of test execution results, with the added responsibility of failure reporting. Failure analysis plays a key role in the estimation of software reliability (dealt with primarily in the next section), the importance of which is emphasised by Cai (1998), and Patel and Ramachandran (2008). Fenton and Ohlson

(2000) have provided an interesting insight into software failures. They found the following through their research:

- The Pareto principle of distribution of faults and failures does actually apply and that a small number of modules contain most of the faults discovered in both pre-release and also in the case of post-release software.
- However it was also discovered that those modules that proved to be the most error prone, pre-release, turned out to be amongst the least error prone, post-release, and vice versa.
- The above could neither be explained by the size nor complexity of the software, nor was there any evidence to suggest that there was a relationship between the size of a software module and fault density.
- There was no evidence to suggest that popular complexity metrics were good predictors of failure. The number of failures discovered in pre-release testing was found to be a multiple of those found in post-release software.

The benefit of recording test failure results, as outlined by Eickelmann and Richardson (1996), is supported by Kuhn et al. (2004). They also state that empirical research into quality and reliability has suggested that there is at least some evidence to suggest that relatively few parameters within software systems are actually responsible for failures. It is suggested that, because we can never know in advance, what interaction is required to trigger all faults in a system, that a more practical alternative to exhaustive testing is to record failure interactions, and the related parameters. A long history of certain failures and associated parameters, could allow the reduction in parameter sets for future test runs, by focussing on combinations of parameters which have previously resulted in failure. The analysis and associated measurement of collected test failure data, is carried out as part of the following, test measurement stage of system testing.

2.4.5 Measurement of System Quality

Eickelmann and Richardson (1996) have made reference to test measurement as including test coverage and test failure analysis. The resulting artefacts are test coverage measures and test failure measures. This is described as supporting the evaluation-oriented period, and enabling the evaluation and improvement of the test

process. The importance of reliability estimation as an indicator of software quality has been previously mentioned as being emphasised by Cai (1998), and Patel and Ramachandran (2008). Horgan and Mathur (1996) have highlighted the importance of reliability estimation is to a software development process, providing organisations with a method of quantifying the level of quality associated with a software product. This method is not without its difficulties however, and the aforementioned authors highlight difficulties associated with the inaccuracy of operational profiles and thus the potential inaccuracy of any estimated reliability. Sommerville (2007) has referred to the difficulties associated with test failure measurement, or reliability measurement:

- *Operational profile uncertainty:* The operational profile may be based on experience with other systems and may not be an accurate reflection of the real use of the system.
- *High costs of test data generation:* It can be expensive to generate large volumes of data required in an operational profile unless the process can be heavily automated.
- *Statistical uncertainty:* when high reliability is specified: You have to generate a statistically significant number of failures to allow accurate reliability measurements. When the software is already reliable, relatively few failures occur and it may be difficult to generate new failures.

Operational profiles have been previously described as an important element of black box testing by numerous authors ((Horgan & Mathur, 1996), (Loveland, Miller, Prewitt, & Shannon, 2005), (Sommerville I. , 2007)). Another concern regarding reliability estimation is highlighted by Tsui and Karam (2007), who stated that software stability is demanded for reliability estimation, and thus any such estimation is usually applied at the completed software stage. Horgan and Mathur (1996) stated that because of the implicit relationship between test case development, and reliability estimation involved with black box testing, that this is not an adequate method of reliability estimation. They develop a methodology to cater for reliability estimation as an iterative software development process, consisting of test execution, fault identification, software modification (there is an assumption of a relatively high level of hardware reliability) and re-testing. This proposed method involved both black box and white box testing.

This view is supported by a model proposed by Littlewood et al. (2002), who in a discussion of a solution to reliability assessment of diverse fault tolerant software based systems, stated that the best way to assess the failure of such systems is to observe under failure at a white box level. A black box approach to testing is considered whereby the probability of failure on demand could then be calculated from the amount of realistic testing performed and the number of failures seen but this model is ruled out, due to the amount of testing and associated costs required for a high PFD (or PFOD) value. Instead the authors investigate a combination of white box testing and a number of inference procedures. These procedures required certain assumptions to be made regarding reliability and were, by the authors own admissions, quite complex to implement.

The model put forward by Horgan and Mathur (1996) suggested incorporating knowledge gained during white box testing into reliability estimation, with the aim of reducing the effect of operational profile errors on reliability estimates. This solution is based on time/structure based software reliability estimation. The authors maintain that a software reliability metric which relates to the probability of software failure within a specified time of operation is a very important and useful metric. This metric can be used to decide whether to release the software or not at any given time. A large number of software reliability models are described as applying to data obtained from working software which has resulted in the accuracy of such models regarding the predicted versus the actual software failure, varying from one project to another. In this particular case the model put forward takes account of the fine structure of the software under development, distinguishing the aforementioned authors' model from other models which may also employ time-domain models. It is also claimed by the authors that structure based models are more likely to provide more accurate reliability estimates that the existing time-domain based models.

Defining T_k as the time at which the kth failure occurs and N_k as the number of test cases used by T_k. E_k is defined as the effort spent in testing:

$$E_k = T_k - T_{k-1}$$ ....in relation to time based testing

and

$$E_k = N_k - N_{k-1}$$ ...in relation to test-case-based models.

Denoting e_i as the effort spent during the *i*th execution of *P* and E_k can be expressed as:

$$E_{\mathrm{k}} = \sum_{i=l_1}^{l_2} e_{\mathrm{i}}$$

Whereby e_{l1} and e_{l2}, respectively, denote the effort spent in the first and last executions of *P* during the *k*th failure interval. The reliability *R* or *P* is defined as the probability of no failure over the entire input domain, *D.*

R = P{*P(d)*is correct for any $d \in D$} ...where *d* is a selected test case from the input domain *D.*

According to Horgan and Mathur, a common assumption made during black box testing is that testing is carried out in accordance with the operational profile. This implies that testers know and make use of the operational profile of the inputs. Knowledge of the operational profile implies knowing what frequency distribution relates to specific test inputs when the software operates in its intended environment. Reliability models put forward by the aforementioned authors, impose test methodologies, with the effect of improving data input to a reliability model. The outcome is a better reliability estimate with predictions being less sensitive to the possible differences between the true operational profile, and its approximation, derived during testing.

Test failure measures

With the verification and validation of failure, which comes as a result of the failure analysis stage, we are in a position to carry out failure measurement. Although recognised as just one aspect of software quality, software reliability is accepted as a key factor since it enables the quantification of software failures (Lyu, 1996). According to ANSI, it is defined as "the probability of failure-free software operation for a specified period of time". Cai (1998) stated that software reliability is the most important software attribute and ranks issues relating to software reliability alongside

those of cost, schedule, and functionality. Similarly, Patel and Ramachandran (2008) rank reliability (2008), as one of the primary indicators of software quality. Sommerville (2007), have stated that software reliability is a complex concept that should always be considered at system level rather than at component level. The reason provided for a adopting a system view is that failure can propagate through a system and affect the operation of other components. The complexity associated with reliability estimation has been emphasised by Littlewood et al. (2002), but the view of adopting a system wide view is argued against by other authors ((Horgan & Mathur, 1996), (Lin, Chou, Lai, Huang, & Chung, 2012)).

Cai (1998) has stated that accompanying the focus on software reliability, are metrics relating to reliability, run reliability, failure intensity and Mean Time To Failure (MTTF). A distinction is made between dynamic software reliability behavior, and static software reliability. Dynamic software reliability is described as being heavily dependent on the operational profile of the software (operational profiles are discussed in more detail in the following test creation section). Identical software systems are stated as possibly demonstrating dramatically different reliability behavior, depending on the operational environments. MMTF is given as an example of dynamic software reliability metric. The role of dynamic software reliability estimation is also emphasized by Littlewood et al. (2002) who stated the importance of being able to estimate the probability of failure per demand (PFD) of safety critical software systems.

In support of both dynamic reliability estimation, and approaching such estimation from a white box perspective, Littlewood et al. (2002) stated that the simplest way to assess the reliability of a system, fault tolerant or otherwise, is to observe failure, whether real or simulated, under operation. Reliability estimation from white box perspective is stated as ignoring the fact that the system is fault-tolerant. Static software reliability, which is independent of software operational profiles, is described as attracting significantly more attention from software development personnel. The number of faults remaining in software is provided as an example of static software reliability metric. In the case of reliability estimation relating to software, Horgan and Mathur (1996) have made reference to the valuable output of failure data, a characteristic of system test which can be used to facilitate this activity. Failure data is

obtained by testing the system against a series of inputs associated with specific test cases. Metrics relating to the estimation of software reliability, referred to by Horgan and Mathur (1996) are:

1. *Probability of failure on demand (POFOD or PFD):* This metric also relates to dynamic software reliability and is most appropriate for systems where services are demanded at unpredictable or at relatively long intervals and where there are serious consequences if the service is not delivered, (Littlewood, Popov, & Strigini, 2002). This can be measure by the number of system failures given the number of requests for system services. The difficulty associated with estimation of this metric is referred to by the aforementioned authors.
2. *Rate of occurrence of failures (ROCOF):* This metric should be used where regular demands are made on system services and where it is important that these services are correctly delivered. This can be measured by the time (or number of transactions) between system failures.
3. *Mean time to failure (MTTF):* This metric also relates to dynamic software reliability and should be used in systems where there are long transactions. That is, where people use the system for a long time. The MMTF should be longer than the average length of each transaction. This can be measured by the time (or number of transactions) between system failures.

Many authors have referred to the use of test information in the estimation of the quality of a software system, and the importance of reliability as a goal of software quality ((Farr, 1996), (Horgan & Mathur, 1996), (Yoo & Harman, 2010), (Lin, Chou, Lai, Huang, & Chung, 2012)). As well as a key tool in the estimation of software quality, reliability prediction can also aid in the identification of optimal test selection and the removal of redundant test cases (refer to section 4.2). This can have a significant impact on the costs associated with the test and overall development process, (Lin, Chou, Lai, Huang, & Chung, 2012). Various authors have employed various methods in reliability prediction, from both a white box and a black box perspective, (Yoo & Harman, 2010).

2.4.6 Management of the Test Environment

Test Management is described by Eickelmann and Richardson (1996) as including support for the complete test environment, including preservation of the test environment state. Desai and Shah (2011) refer to this stage as involving a graphical layout of the test architecture, the test equipment, quantities and descriptions with possible accommodation for multiple test environments catering for test scalability and with a focus on test time reduction. Test architecture, which forms an important part of this stage, is described by Walter and Grabowski (1999) as being a combination of:

- *Test equipment.*
- *The actual system under test.*
- *All interconnectivity between elements of the system under test.*

The important of test equipment is referred to by author such as Loveland et al. (2005), who state that the execution of many test activities by system testers and in particular performance testers (which may be focussing on load or stress testing), could not be performed without the availability of such tools. Tsui and Karam (2007) have made reference to the complexity associated with the software testing task, and the many activities of software test involving test methodologies, techniques, tools, and resources, necessary in order to achieve required goals. Eickelmann and Richardson (1996) have stated that the test architecture facilitates the test environment and the previously referred to test functions, namely:

- Test execution
- Test development
- Test failure analysis
- Test measurement
- Test management
- Test planning

It is stated that the same qualities which are important to software, are also important to a software test environment, namely *correctness, reliability, efficiency, integration,*

usability, *maintainability*, *flexibility*, *testability*, *portability*, *reusability* and *interoperability*. Difficulties involved in facilitating the replication of customer environments are described by Loveland et al. (2005). The size of customer environments has been referred to as a particularly difficult thing to replicate, which is important in terms of scalable tests, Desai and Shah (2011). Size combined with other customer specific characteristics such as distributed systems with interconnecting cables can be very difficult and costly to implement. Other difficulties associated with this stage include the potential heterogeneous nature of customer environments whereby it is very highly likely that there are significant differences between different customer environments. These environmental differences should therefore be accommodated in a test environment where possible. Along with the difficulties associated with the practical implementation of customer environments, the lack of understanding of customer environments, which may exist in both development and test teams, is also highlighted as a potential issue for test management. This may cause both the non-recognition of customer usage, as well as the dismissal of valid usage as unrealistic. The purpose of this stage is to facilitate the test environment to enable test execution. Test execution and associated issues is discussed in more detail in the forthcoming section.

This section has emphasised the important role which test measurement plays in any test process. This concludes an overview of the previously identified functions, identified by Eickelmann and Richardson (1996) and Desai and Shah (2011), namely *test planning*, *test development*, *test execution*, *test failure analysis*, *test measurement*, and *test management*. Also discussed in this section was testing from a perspective of developers, testers and users, as well as focusing on testing from a perspective of *test objectives*, *test approach* and *test architecture*, which is in keeping with the views of Walter and Grabowski (1999). The following chapter provides a greater understanding of the types of complexity which potentially affect software development environments.

2.5 Concluding Analysis of Software Development Processes and System Testing

As part of an overview of software development methodologies, the views of Rajagopalan (2014) were discussed. Views such as those expressed by Rajogoplan, have helped explain the movement from traditional software development methodologies, to increasingly agile methodologies. He has stated that concerns over quality and the future maintenance of software, led to the widespread adoption of traditional methodologies, such as Royce's waterfall model (Royce, 1970). The necessity of a more flexible approach to software development and the emphasis of a "practice over process" approach is something which led to the development and adoption of more agile approaches to software development. Highsmith and Cockburn (2001) and Chau (2004) have held the view that changing customer requirements should be embraced, and that models that enable such a rapid software change (similar to those advocated from an agile approach) are superior. The focus on the software development process characteristic of flexibility, particularly by agile development methodologies, has resulted in a concentration on certain aspects of software testing. Crispin and Gregory (2009) referred to the emphasis on agile as being reflected through software testing being defined by the business experts' desired features and functionality, and not generally by tests which critique the product.

As part of a software development overview in section 2.3 of this chapter, fundamental aspects of development processes were outlined which are common across different approaches to software development i.e. irrespective of whether a traditional or agile approach to software development is adopted. These were in keeping with the work of Huo et al. (2004), and identified as:

1. *Software specification and design:* The functionality and constraints associated with the software must be defined. This may take the form of requirements definition and software and system designs or alternatively approaches such as user stories, system metaphors, architectural spikes, and release planning.
2. *Software implementation:* In line with the requirements, goals and designs, the software must be produced. This can be a planned iterative development process, or a planned linear development process.
3. *Software verification and validation:* The software must be validated to ensure it acts in accordance with customer requirements or standards. Code

> verification and validation can take the form of static checks such as code reviews, inspections, and peer programming, or dynamic approaches such as software testing in the form of unit and system testing. Validation can also take the form of customer feedback and acceptance testing.

Tsui and Karam (2007) highlighted several methods which can be used for detection of errors in programs, both from a static point of view (verification and validation of non-running code e.g. via code reviews), and from a dynamic point of view (verification and validation of running code):

- *Testing* involves executing the software in a controlled environment and verifying that the output is correct.
- *Inspections and reviews*, which can be applied to programs or relevant documentation. These generally involve more than one participant, in addition to the document or program creator. These are described as being labour intensive, but an extremely effective method of finding errors.
- *Formal Methods* involve mathematical techniques which are used to prove that a program is correct.
- *Static analysis* involves analysing the static structure of a program or relevant documentation. Usually automated, this method can detect errors or error-prone conditions.

Such methods are common in both traditional and agile software development environments ((Huo, Verner, Zhiu, & Bahar, 2004)). As referred to in the introductory section, Sommerville (2007) has emphasised that techniques such as software inspections, automated source code analysis, and formal verification, can only verify that a program is in accordance with the specifications, and cannot demonstrate whether the software is operationally useful (this view is endorsed by Delahaye et al. (2013)). Software testing, a dynamic validation and verification techniques, has been identified as an important part of the software development process ((Eickelmann & Richardson, 1996), (Cai & Card, 2008), (Desai & Shah, 2011), (Kochhar, Bissyand, Lo, & Jiang, 2013)). It is described as being the foremost method for software validation and verification, checking properties of the software such as performance and reliability (Holzworth, Huth, & deVoil, 2011). The importance of software testing

is also emphasised by other authors ((Wegener, Baresel, & Sthamer, 2001), (En-Nouaary, 1998), (Mattiello-Francisco, Martins, Cavalli, & Yano, 2011), (Yin & Ding, 2012)).

Wegener et al. (2001), Mattiello-Francisco et al. (2011) and Yin and Ding (2012) emphasised the merits of a structured approach to software testing, in terms of effectiveness and efficiency, over adopting an ad-hoc approach. A structure to testing has been provided by Eickelmann and Richardson (1996), who has highlighted key functions which software test environments have evolved to include, over a period of time:

1. ***Test Execution*** includes the execution of the instrumented source code and recording of execution traces. The output of this stage includes test output results, test execution traces, and test status.
2. ***Test Development*** is essentially the development of a *test approach*, which includes the specification and implementation of a test configuration. The output of this stage is the test suites and the individual test cases, test input criteria, test documentation, and test adequacy criteria.
3. ***Test Failure Analysis*** includes behavior verification and documentation. The output of this stage includes recording of test results (such as pass or fail) and test failure reporting.
4. ***Test Measurement*** includes test coverage measurement and analysis. Source code is described a typical instrument used to collect execution traces. Executed test runs have associated with them test coverage measures and test failure measures.
5. ***Test Management*** includes support for the complete test infrastructure along with test execution state preservation. The test process may require a repository for the test infrastructure.
6. ***Test Planning*** includes the development of a plan relating to test case development. This is described as including the foundations for *test objectives*. This involves detailing the features of the system to be tested, risk assessment issues, organizational training needs, required and available resources, development of a comprehensive test strategy, reconciling required and available resource and staffing requirements, roles and responsibility

allocations, and overall schedule. Development of a *test architecture* which outlines the required and available resources would also be carried out at this stage.

Fundamentally, the model proposed by Desai and Shah (2011) relating to the different functions of software test, is similar to that highlighted above, with the slight difference of an emphasis on a *test environment preparation* stage, as opposed to a *test management* stage. Accepting that test management is an ongoing activity, which may be invoked at the start of projects also, and test case planning is carried out at the beginning of a project, the following order is proposed as the standard execution order of the aforementioned test related functions:

1. ***Test Planning***
2. ***Test Development***
3. ***Test Execution***
4. ***Test Failure Analysis***
5. ***Test Measurement***
6. ***Test Management***

Covered in figure 2.9 are the important key aspects of test objectives, test approach, and test architecture, as referred to by Walter and Grabowski (1999).

Test Functions and Considerations (Walter and Grabowski (1999)).

Test Planning	Test Case Development	Test Execution	Test Fault Analysis	Test Measurement	Test Management
Test objectives (functional v non-functional). Understanding features to be tested. Risk assessment. Facilitating training requirements. Balancing necessary vs available resources (both human and technical). Test strategy (test selection, minimisation and prioritisation). Roles and responsibility. Schedule development.	Implementation of a test approach i.e. a complete test configuration (facilitating white box or black box). Development of test suites.	Test Execution against system under test. Test artefact recording i.e. test output results, test traces, test status.	Test result verification. Test result analysis and documentation (pass/fail, test coverage).	Test coverage measurement. Test failure measurement.	Consideration of the test architecture and test environment preservation. Maintenance of test resource repository (necessary in the case of an automated test process).

Figure 2.9: Test functions and considerations.

As part of verification and validation, the importance of software testing to the development process has been dealt with in this chapter. The next chapter addresses the two core elements of this research:

1. *Complexity associated with the task of system testing.*
2. *The relationship between system test complexity and tacit knowledge.*

The strong relationship between complexity associated with aspects of the software development process, and knowledge, has been highlighted by numerous authors, from a general software development perspective ((Staats, Valentine, & Edmondson, 2010), (Lu, Xiang, & Wang, 2011), (Wang, Huang, & Yang, 2012)), and specifically from a geographically distributed development team perspective (Espinosa, Slaughter, Kraut, & Herbsleb, 2007). In the case of Lu et al., the complexity of information systems development is acknowledged, as is the necessity of knowledge sharing, identified as an important factor in the development of information systems. Staats et al. (2010), in their research carried out at Wipro Technologies, relating to the use of knowledge repositories, have investigated how the use of knowledge affects performance. As a result of this research, the importance of the distribution of knowledge amongst team members is emphasised, particularly in the case of complex tasks.

The strong relationship between system testing and knowledge has been emphasised by Talby et al. (2006), and Desai and Shah (2011). Talby et al. referred to the importance of knowledge to independent test teams, and raised concerns regarding the availability of knowledge under certain geographical settings. Similar difficulties have been highlighted by others ((Chau & Maurer, 2004), (Lee, Delone, & Espinosa, 2006)). Cataldo and Ehrlich (2012) have made reference to the lack of existing research, which examines the communication structures facilitating the transfer of knowledge, something which is considered key in software development processes, and also to the overall achievement of software development goals, such as productivity, and quality. The importance of tacit knowledge to software testing has been emphasised by Andrade et al. (2013), and a case for further research into the area of tacit knowledge and the role which it plays in software development processes has been made by Ryan and O'Connor (2009), and Dingsøyr and Šmite (2014), who have emphasised the need for a greater understanding of this particular topic.

3 A Review of Software System Test Complexity and Tacit Knowledge

The goal of this research, much in keeping with the views of Casti & Karlqvist (1986), is an attempt to reduce the effects of complexity through understanding its characteristics, influences and effects. As an introduction to complexity relating to software and in line with the views of Brooks (1995), software complexity can be viewed from two different perspectives:

3. *Complexity inherent in software.*
4. *Complexity associated with the process of software development*.

The topic of inherent complexity is dealt with in significant detail by Perrow (1984), who referred to the inherent complexity associated with technological systems in general, and the potential negative consequences of such complexity. Complexity is stated as an inevitable consequence of some system designs, necessary in order to achieve the intended goals of the system, often providing efficiency through system characteristics such as multifunctional components. The concept of inherent complexity associated with software systems is endorsed by other authors ((Mumford, 1983), (Brooks F. , 1995), (Lehman, 1996), (Lyytinen, Mathiassen, & Ropponen, 1998), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007), and (de Silva & Balasubramaniam, 2012)). The second perspective of complexity, as outlined by Brooks, relating to complexity associated with the process of software development, is of significant relevance to this research because of the interest in complexity associated with the task of system testing. Espinosa et al. (2007), after research relating to distributed software development teams, have stated that complexity varies greatly depending on the characteristics of the software task, like size and structure, and on environmental conditions, such as team size and geographic dispersion. Lee et al. (2013) emphasised the importance of process standardization, process rigor, and process agility, in dealing with such complexity.

Lu et al. (2011) have acknowledged the general complexity of information systems development, and the necessity of knowledge sharing, in any effort to mitigate the effects of such complexity. In line with the views of Lu et al. (2011), Rus et al. (2001) and Pee et al. (2010), have also highlighted the increasingly important role which

knowledge plays in the software development process, and state it is necessary to leverage individual knowledge at both a project and organisational level, so as to ensure optimal software development. The topic of tacit knowledge is strongly linked to the human aspects of software development, as opposed to technological aspects ((Faraj & Sproull, 2000), (Ryan & O'Connor, 2009)). Ryan and O'Connor (2009) have emphasised this perspective in questioning the contribution of technological solutions to the performance of successful projects, instead highlighting the importance of such human factors. The importance of effective plans, good communication and clear goals, are specifically referred to, and a link is provided between the role of tacit knowledge, and the success of software development teams. The effective utilisation of tacit knowledge is stated as demanding a structured knowledge management approach ((Rus.I, Lindvall.M, & Sinha, 2001), (Desai & Shah, 2011)). Even though such an approach to knowledge management is stated as demanding time and effort, at both an individual and organisational level, if applied in the case of software testing, it is stated as eventually leading to a reduction in time, cost, and effort. This is stated as being applicable for any software testing which may be carried out in the case of future projects (Desai & Shah, 2011). The role of knowledge in software development forms an important part of further discussions in this chapter, with particular emphasis being placed on the role of tacit knowledge.

There have been recognised benefits associated with applying socio-technical models in helping to understand the effect of information systems in organisations ((Lyytinen, Mathiassen, & Ropponen, 1998), (Vidgen & Madsen, 2003), (Herbsleb, 2007), (Sommerville I. , 2007), (Lu, Xiang, & Wang, 2011), (Sommerville, et al., 2012), (Davis, Challenger, Jayewardene, & Clegg, 2013)).The socio-technical model, as outlined by Mumford (1983), in figure 3.1, (based on the original work of Leavitt (1954)), provided a useful tool when highlighting the organisational, human, task, and technological aspects of software development, as used in the aforementioned discussion relating to the importance of knowledge sharing in systems development.

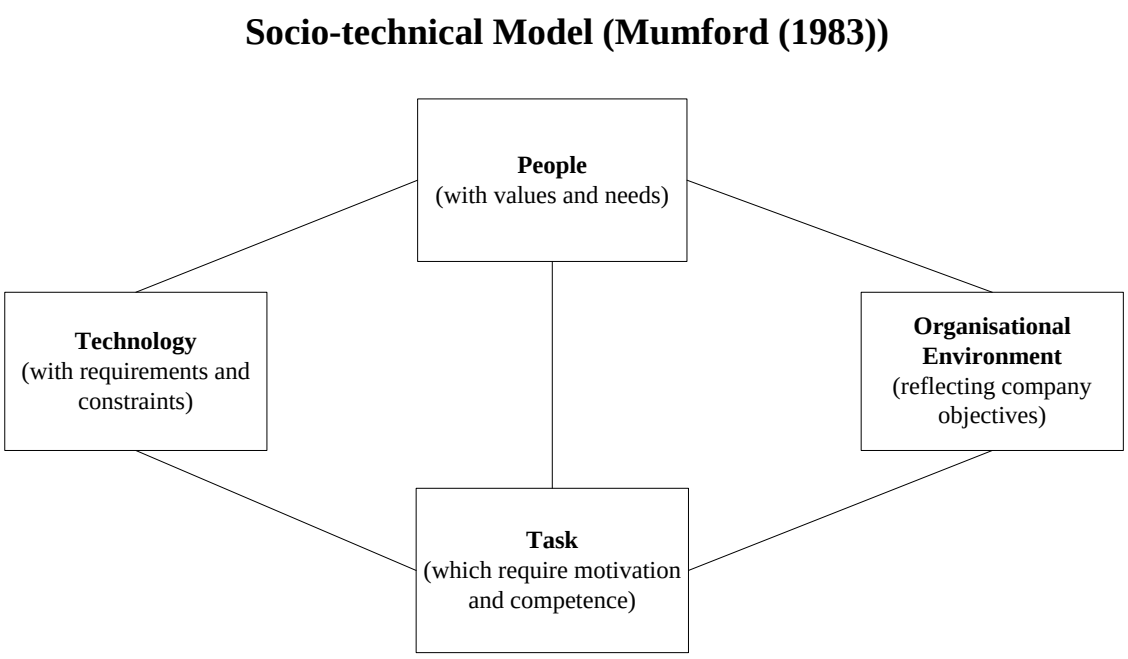


Figure 3.1: Socio-technical of Information Systems.

One criticism of the original model (Leavitt, 1964), was its static nature and lack of reference to environment, something which Mumford included when applying the model to the area of software development, (Mumford, 1983). The reference to organisational environment instead of referencing organisational structure is something which other authors have also taken account of (Lyytinen, Mathiassen, & Ropponen, 1998). The model views organisations as comprising of four interacting components: tasks (requiring motivation and competences), organisational environment (reflecting company objectives), people (with values and needs) and technology (with requirements and constraints). The aforementioned model, as proposed by Leavitt, suggests that the four aforementioned components are strongly related, and that a change in one has an effect, whether planned or unplanned, on the other components. The framework also proposes that these components are continuously changing and interacting due to environmental influences and those variations are both constant and inevitable.

This socio-technical model is applied at various stages throughout this chapter. The application of the model is aimed at providing a consistent socio-technical link through

discussions regarding system test complexity and the role of tacit knowledge. The increasing importance of viewing system testing from a socio-technical perspective has been made by Mantyla et al. (2012). Though commonly applied in the case of system design, to help provide an understanding of the potential effects of systems on organisations (Sommerville I. , 2007), views have been expressed relating to the benefits of applying a socio-technical models to a wider context of issues involving complex systems (Davis, Challenger, Jayewardene, & Clegg, 2013). The following sections provide an insight into the relationship between complexity and the task of system testing, with a particular interest also being shown for the relationship between tacit knowledge and system testing.

3.1 The Influence of Complexity on Software Testing

As stated in the introductory section, the identification of complexity associated with the task of system testing is a key element of this particular research. Steinmann (1976) held the view that complexity equated to the absolute amount of information involved in a task, the internal consistency of that information, and the variability and diversity of that information. In relation to the task of system testing, Debbarma et al. (2011) have argued that there has been increasing complexity, along with the increasing size and performance demands of software systems, all of which demands more effective software testing. Other difficulties associated with the role of the software tester have been highlighted by Loveland et al. (2005), who infer that the role of software testers have progressively become more demanding, from not only ensuring that among the defects found are all the defects that would disrupt real working environments, but to also validating other system characteristics through specific testing, such as performance and system recovery testing. Tsui and Karam (2007) have adopted a similar point of view, highlighting the general complexity associated with the task of software testing, and the many activities of software testing, involving test methodologies, techniques, tools, and resources, which are commonly used in order to achieve required goals. Baig and Khan (2010) have taken a slightly different perspective, focussing on the goals of system testing, stating that significant difficulty and complexity associated with testing, stems from the question of how to

carry out testing more efficiently. The aforementioned authors identify the goal of test time reduction, without impacting the software testing goals of correctness, completeness, and quality, as being an important source of complexity.

The difficulty of providing test coverage for large or complex systems has been highlighted (Zheng, Alager, & Ormandjieva, 2008), (Lin, Chou, Lai, Huang, & Chung, 2012), (Ferrer, Chicano, & Alba, 2013)). In keeping with this view, Myers (1979) has made reference, not alone to the difficulty and complexity associated with providing adequate test coverage, but the impracticalities with providing exhaustive test coverage for software systems in general. Subsequent sections deal with different aspects of complexity, associated with the process of software development, an area in which considerable research has been carried out, identifying complexity from a number of different perspectives, such as general task complexity ((Wood, 1986), (Campbell, 1988), (McKeen, Guimaraes, & and Wetherbe, 1994), (Li, et al., 2011)), complexity associated with specific tasks such as system deployment ((Ribbers & Schoo, 2002), team complexity (Espinosa, Slaughter, Kraut, & Herbsleb, 2007)), and project complexity ((Lyytinen, Mathiassen, & Ropponen, 1998), (Pee, Kankanhalli, & Kim, 2010)).

This research is primarily concerned with complexity associated with the task of system testing. The importance of task complexity is emphasised by authors such as Li et al. (2011), who highlight such complexity as a task characteristic which has a significant effect on task performance. Through their analysis of literature relating to the topic of task complexity, Li et al. (2011) have identified two general perspectives which have been adopted in relation to task complexity:

1. An *objective* perspective, whereby task complexity is a characteristic of the task.
2. A *subjective* perspective, whereby task complexity is complexity as perceived from the task doer.

Wood (1986) and Campbell (1988) are stated as referring to objective complexity. In line with the views of Campbell, Li et al. (2011) defined objective task complexity as implying “an increase in information load, information diversity, or a change in the

rate of information". Subjective task complexity is described as the degree of complexity of a task, from the perspective of the task executer. The link between the task of system testing, and project complexity, is provided by Pee et al. (2010), who highlighted the relationship between task performance and project complexity, through their research relating to knowledge sharing in information systems development.

3.1.1 Software Project Complexity

A discussion of complexity from a perspective of the overall project has been taken by a number of authors ((Wood, 1986), (Baccarini, 1996), (Xia & Lee, 2005), (Williams, 1999), (Pee, Kankanhalli, & Kim, 2010), (Açıkgöz, Günsel, Bayyurt, & Kuzey, 2013)). Wood (1986) has stated the greater the number of software changes in a particular software project, the more complex that software project inevitably is. Much in keeping with that view, Turner and Cochrane (1993) have suggested that project complexity is relative to the extent to which project goals are poorly defined, and are subject to future changes. Baccarini (1996) have defined project complexity in terms of the number of varied elements, and interdependency between those elements. Project complexity is stated as comprising of organisational complexity and technological complexity. Organisational complexity is defined as encompassing relationships, hierarchical levels, formal organisational units and specialisations. Technological complexity is defined as encompassing inputs, outputs, tasks and technologies. A similar classification theme is followed by Williams (1999), who identified structural complexity and uncertainty-based complexity. He contended that a complete picture of project complexity includes not only structural complexity originating from the underlying structure of the project but also uncertainty-based complexity originating from the changes in the project environment. The author maintained that the distinction between structural and uncertainty based complexity is important, because he states that organisations tend to deal well with structural complexity, but do not tend to be sufficiently equipped to deal with uncertainty based complexity. Shenhar and Dvir (1996) have suggested that the uncertainty-based complexity is based on the level of technological uncertainty at the initial stage of the project (Shenhar & Dvir, 1996). Xia and Lee (2005) suggested that technological complexity demands a more dynamic approach.

Comprehensive analysis of available research in the area of information system project complexity has been carried out by Xia and Lee (2005). At one level the framework differentiates between structural complexity and dynamic complexity, and on another level the framework differentiates between organisational and technological complexity. Structural complexity is defined as variety, multiplicity, and differentiation of project elements and the interdependency, interaction, coordination and integration of project elements. Dynamic complexity is defined as the uncertainty, ambiguity, variability and dynamism, which are caused by changes in organisational and technological project environments. On another level, a differentiation is made between organisational complexity and technological complexity. Organisational complexity is defined as the complexity of organisation environments surrounding a project. This is described as including stakeholders such as user groups, senior management, project teams, contractors, vendors as well as organisational structures and business processes. Technological complexity is defined as involving the technological environment of the information systems development project. This may include the technological platform, design techniques and computing languages, development methodologies, and system integration (McKeen (1994)).

The four complexity dimensions of information system development projects identified by Xia and Lee (2005) are:

- *Structural organisational complexity:* the multiplicity and interdependency of organisational elements of an information systems development project.
- *Structural IT complexity*: the multiplicity and interdependency of technological elements of an information systems development project.
- *Dynamic organisational complexity*: the rate and pattern of changes in the information systems development project organisational environments, including changes in user needs, business processes, and organisational structures.
- *Dynamic IT complexity:* The rate and pattern of changes in the IT environment of an information systems development project, including changes in IT infrastructure, architecture and software development tools.

Changes may occur as a result of the stochastic nature of the environment or a lack of information and knowledge. Dynamic complexity is described as being increasingly relevant because both business and IT environments are changing with unprecedented pace.

A significant distinction between the frameworks detailed by Williams (1999) and the framework highlighted by Xia and Lee (2005) is the prominence of what Xia and Lee claimed is that dynamic complexity associated with technology and organisational environment. Xia and Lee also identified the specific characteristics of each complexity dimension which they have concluded from literature.

So, in line with the view of Baccarini (1966), project complexity appears to touch on all aspects of the socio-technical model, as detailed in figure 3.2.

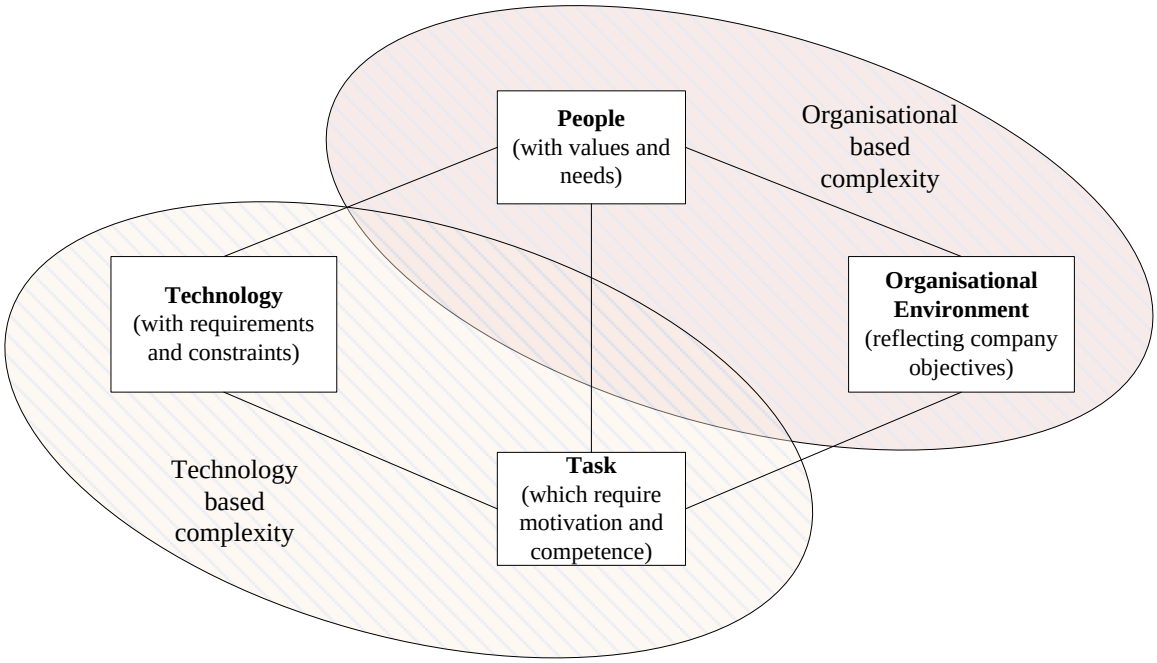


Figure 3.2: Project Complexity from a Socio-Technical Perspective.

Referenced as part of the previous discussion on project complexity, is the topic of task complexity. The next section will focus specifically on the topic of inherent complexity. The significance of further research in the area of inherent complexity,

and indeed the importance of the socio-technical aspects to future research, which has been highlighted by Sommerville et al. (2012).

3.1.2 Inherent Software Complexity

This characteristic of inherent complexity associated with software and software systems in particular is something which numerous authors have made reference to ((Mumford, 1983), (Brooks F. , 1995), (Lehman, 1996), (Lyytinen, Mathiassen, & Ropponen, 1998), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007), (de Silva & Balasubramaniam, 2012)). Brooks (1986) stated that computers are described as more complex than most things people build but software is described as having orders-of-magnitude more states than computers. Analysing such complexity, and in line with the thoughts of Aristotle, Brooks makes the following distinction between essential complexity and accidental complexity associated with software engineering:

- *Difficulties associated with the nature of software are referred to as essentially complex.*
- *Difficulties associated with software production are referred to as being accidentally complex.*

Conceptual constructs associated with software are described as being essentially complex, affecting the specification, design and test of software systems. Similar views have been expressed by de Silva and Balasubramaniam (2012), who recognised the negative consequences associated with inherent software complexity in terms of maintenance and modification. Such complexity is stated as making it harder to understand and change software designs. This leads developers to make engineering decisions which could damage the architectural integrity of the system. The modification of software is described as extremely complex, because software elements are described as inevitably interacting with each other, thereby increasing the whole complexity of the system ((Brooks F. P., 1986), (Bhattacharya, Iliofotou, Neamtiu, & Faloutsos, 2012)).

The effects of the tight coupling of software components

Some authors have made reference to naturally increasing complexity associated with evolving software systems (referred to as E-Type systems by Lehman), unless deliberate attempts are made to reduce such complexity ((Lehman, 1996), (de Silva & Balasubramaniam, 2012), (Bhattacharya, Iliofotou, Neamtiu, & Faloutsos, 2012)). Bhattacharya et al. (2010) have referred to the difficulties, complexity, and costs associated with ensuring the reliability of evolving software systems. Similar views have been expressed by Espinosa et al. (2007), who have also referred to the complexity associated with the modification of software, due to the tight coupling of software module interdependencies. The relationship between tight coupling of system components and complexity is a topic which has been analysed by Perrow (1984). The aforementioned authors provide some possible reasons for tight coupling, whereby components are interdependent and the performance of one tightly coupled component has a direct effect on the performance of another tightly coupled component. Pressures due to system timing are described as possibly requiring the tight coupling of components, in order to achieve performance, quality, or efficiency goals.

Similar views referencing the trade-off between performance improvements and complexity are echoed by de Silva and Balasubramaniam (2012), with complexity identified as a natural characteristic of many system designs, introduced through attempts to accommodate new user requirements and maintain acceptable levels of performance, often carried out in order to prevent software becoming obsolete too soon. Perrow offers a reason as to why software systems are regularly so complex. In some cases it is argued that complexity is a natural consequence of some system designs because the knowledge or ability does not exist to allow the system to be designed as a linear system with limited interaction between system components. It is argued that the goals of efficiency and performance in some system designs, which regularly involve the presence of multi-functional or multi-mode components, is a major contributor to complexity.

The aforementioned section covered inherent software complexity, which can be considered as relating to technology, figure 3.3.

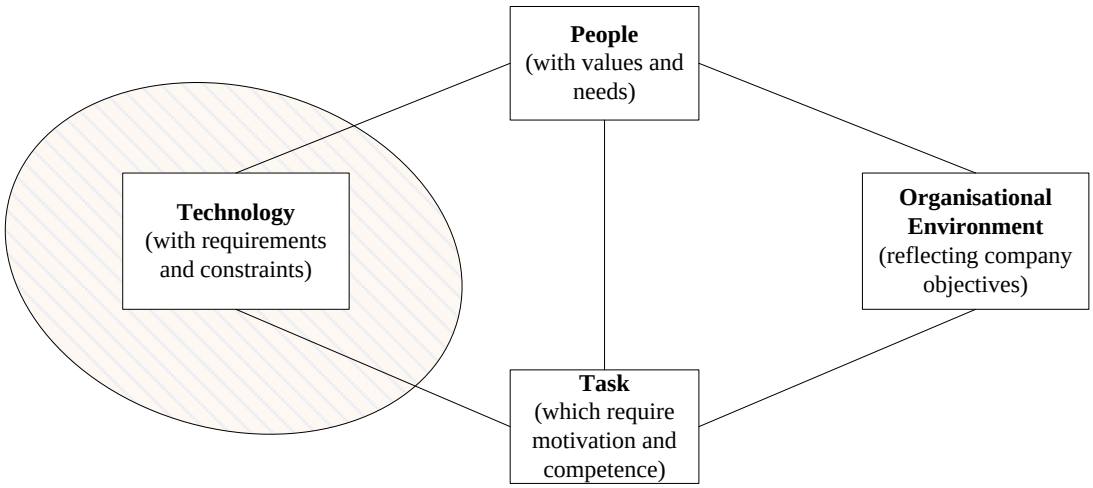


Figure 3.3: Inherent Complexity from a Socio-Technical Perspective.

The following section discusses task complexity, the importance of which has been emphasised by Tsui and Iriele (2011).

3.1.3 Software Task Complexity

Akman et al. (2011) described the software development process as being an error-prone, time-consuming, and labour intensive activity, which can involve considerable complexity. Complexity associated with software testing, an important aspect of the development process, is something which has been highlighted by numerous authors ((Yeates, Shields, & Helmy, 1994) (Zheng, Alager, & Ormandjieva, 2008), (Debbarma, Singh, Shrivastava, & Mishra, 2011)). Yeates et al. (1994) have referred to complexity as being inherent in testing, whereas Akman et al. (2011) have maintained that complexity associated with code written in an increasingly complex manner, can lead to increased complexity in software testing. Loveland et al. (2005) and Martin (2007) have argued that that an imbalance exists, between the advancements made from a software development perspective and from a software testing perspective. They state that while advancements have been made to tools and methodologies associated with the development process, that not nearly the same improvements have been made in relation to software testing tools, to aid the identification of software faults.

In contrast to the aforementioned views, Andrade et al. (2013) have expressed the view that there have indeed been advancements in software testing models, with testing techniques, such as devised by Myers (1979), having been added to by new testing frameworks and techniques. *Model-based testing* and *agile testing* were provided as examples of frameworks, along with examples of new testing techniques, such as *machine learning techniques, adaptive random techniques* etc. It is stated that such advancements, combined with the application of software to new domains and new development models, makes software testing knowledge more intensive and increasingly complex. Tsui and Iriele (2011) have maintained that complexity associated with the software testing relates to one of the sub-tasks of *test case development, test environment setup, test execution and recording* and *test result analysis*. Of the aforementioned tasks, test case development is described as possibly the most challenging and time consuming.

Research in the area of task complexity is stated as having been conducted from a *subjective complexity* or *objective complexity* perspective, (Li, et al., 2011). Objective task complexity is stated as being a characteristic of the task, whereas subjective task complexity is based on the perception of the task executer. A general perspective of task complexity has been adopted by numerous authors ((Wood, 1986), (Campbell, 1988), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007)). An interesting analysis of task complexity highlighting the effects on task accomplishments has been provided by Campbell (1988). This framework presents complexity as having a positive relationship to the following four characteristics:

- *When multiple potential paths to successful goal attainment exist*. Multiple paths lead to increased complexity when multiple paths exist as potential possibilities, but not all lead to successful goal attainment, alternatively when there is efficiency criterion embedded in the task and paths must be evaluated against such criteria. Multiple paths decrease complexity when multiple paths exist, they all lead to goal attainment, and efficiency criteria associated with path evaluation is not relevant.
- *When multiple desired outcomes are required*. Campbell describes it as thinking of each outcome as a task dimension and that complexity increases

with an increase in the number of different dimensions being considered as information processing demands increase. Again the exception being that if all outcomes are positively related, then the degree of complexity reduces.

- *When there exists conflicting interdependence among paths*. Complexity can occur because of a negative relationship amongst desirable outcomes. If achieving one outcome conflicts with achieving another desired outcome, complexity increases. Typically the activities that increase quality preclude the activities leading to quantity. Campbell gives the example of a previous situation, whereby processing had to increase, but that associated labour costs had to decrease and that one objective conflicted directly with the actions of the other.
- *When the connection between path activities and desired outcomes cannot be established with any certainty*. If probabilistic linkages exist, information load is affected i.e. potential paths cannot be eliminated quickly, and diversity is impacted i.e. different action-outcome activities must be evaluated. Uncertainty can also increase complexity through increasing the potential pool of paths to a desired outcome. If such uncertainty exists then the existence of a more effective path must be considered.

Wood (1986) has taken a similar approach to Campbell in that the focus is also on task complexity. Where the views differ, is that Wood (1986) has defined three types of task complexity: *component, coordinative* and *dynamic*. These take into account the quality of task instruction and the changing states of task environments, as well as task execution. Component complexity has been defined as relating to the number of distinct tasks which must be executed in the performance of the task, and the amount of information that must be processed in the performance of those particular acts. Coordinative complexity refers to the nature of the relationships between task inputs and outputs. The form and strength of the relationships between task information, execution, and products, are all defined as aspects of coordinative complexity. Dynamic complexity is caused by changes in the state of the task environment.

Campbell has also identified associative characteristics which are often linked to task complexity such as *lack of structure*, *ambiguity* and *difficulty*. He states that these

require special attention because their relationship to objective task complexity is not straight forward. Poor structure, ambiguity, and difficulty, may also be a consequence of basic task characteristics i.e. tasks which have multiple paths which are imprecisely linked to several desired, but have conflicting outcomes, are likely to be unstructured, and may be difficult and ambiguous. However tasks may be unstructured, ambiguous, and difficult, for reasons other than the characteristics of the task itself. Incomplete training of what would be generally perceived as a straightforward task could be one example of such a situation. The factors which make the task complex are external to the task itself but serve to make the task complex. Campbell has made the distinction between the two, stating that certain tasks may be difficult (require significant effort) but not necessarily complex, and certain other tasks can be difficult because they are complex. There is also the point made that task difficulty is subjective, with a dependence on one's ability.

An important distinction is made between task types via the following task complexity classification:

- *Simple tasks* – appear to contain no task complexity characteristics.
- *Decision tasks* – a common task here involves choosing or discovering an outcome that optimally achieves multiple desired end-states. These tasks normally involve selection of the best alternative from many possibilities. Task types may be distinguished within this category by interdependence among outcomes and by either the absence or presence of uncertainty.
- *Judgement tasks* – these tasks require the individual undertaking the task to first consider and integrate diverse sources of information and subsequently to make a judgement or prediction about the likelihood of some future event. These types of tasks are based on inconsistent or contradictory information and may thus require deeper analysis, prioritisation and assimilation of information prior to any judgement taking place. Examples provided relate to intelligence analysis, stock market analysis etc.
- *Problem tasks* – such tasks are defined as having a common characteristic of multiple paths, leading to a well specified, desirable outcome. These tasks involve finding the best way to achieve the desired outcome. They have been labelled problematic because the tasks differ in terms of the paths, relationship

to each other, and the desirable outcome. Examples of such tasks are given as check problems, anagrams, and jigsaw puzzles etc.

- *Fuzzy tasks* – these tasks are labelled so because they are described as having a common characteristic of having multiple desired end-states, and multiple ways of attaining each of the desired outcomes. An example here is given as involving the manufacture of a new product which included several innovative and attractive characteristics with each characteristic (multiple outcomes) attainable through different production methods (multiple paths).

A slightly different perspective has been provided by McKeen et al. (1994), who focus on information system development complexity, with a distinction being made between task complexity and system complexity, as opposed to task and team complexity in the case of Espinosa (2007). Task complexity is defined in terms of ambiguity surrounding the users understanding of the task. In the context of the research carried out by McKeen et al., system complexity is defined in terms of the development project. Tait and Vessey (1988) have taken a similar view to McKeen et al. defining system complexity in terms of the difficulty in determining the information requirements of the system, the complexity of processing, and the complexity of the overall system design. Meyer and Curley (1991) have defined technology complexity of an expert system, taking into account the diversity of technologies used, database intensity, and integration effort.

A specific task perspective has been adopted by Ribbers and Schoo (2002) who focus on system deployment, and recognising three dimensions of system implementation complexity. The first dimension is *variety*, which is related to the number of project elements involved, such as the number of sites affected by a system implementation. The second dimension is *variability*, which relates to project goal and scope. The third dimension, *integration*, focuses on the coordination of various project elements.

The following section provides a brief discussion on task and team characteristics which can influence task complexity.

3.1.3.1 Dealing with Complex Tasks

Espinosa et al. (2007) has referred to the complexity associated with software tasks in general relating to distributed software development teams, stating that this complexity varies greatly depending on the characteristics of the software task itself, like size and structure, and on environmental conditions such as team size and geographic dispersion. They investigated the effect on team performance, by software tasks and team familiarity, and software tasks and team complexity. Task complexity was simply defined as relating to the magnitude and structure of tasks. The authors investigated whether complexity increases as tasks are larger or structurally more complex (the authors took the number of lines of code affected for task complexity and the number of modules affected for structural complexity). Team complexity was defined simply as relating to environmental conditions such as team coordination, and was gauged by complexity increases when teams are larger or more geographically dispersed (something which is becoming more common (Lee, Espinosa, & DeLone, 2013)). The effect of team sizes on complexity associated with software development tasks has been acknowledged by Akman et al. (2011).

The Role of Task Familiarity

The conclusion reached regarding task familiarity, was that team performance was not affected by task size (number of lines of code added, deleted, or updated). As the size of software tasks increased, software development time increases, and conversely, as task familiarity increases, software development time decreases proportionally but in such cases, no dramatic productivity improvements were attributed to task familiarity. A second view expressed by the authors, was that dramatic productivity improvements are possible in more structurally complex tasks (complexity was defined by the number of modules affected by a particular "modification request" which is dealt with by a developer) through task familiarity alone. This would appear to be supported by Banker and Slaughter (2000) who have stated that task familiarity is increasingly important in larger software tasks, because relevant sections of software areas can be identified more easily, due to a more detailed knowledge of the software product.

Brooks (1995) has expressed a slightly contrasting view, stating that there is inherent complexity in software tasks which is irreducible as software becomes increasingly complex. As this inherent complexity increases, the addition of developer experience is stating as having a negligible effect on such complexity. The aforementioned views would also appear to be at odds with Chau and Maurer (2004), who found that task familiarity helped reduce task completion time for tasks with lower structural complexity only, but that dramatic productivity improvements do not appear possible for more structurally complex tasks, at least not through task familiarity alone. Going some way towards reconciling the views of Espinosa et al. and Brooks, it was suggested that the benefit of task familiarity may be dependent on the source of complexity, rather than the level of complexity.

The Role of Team Familiarity

The relationship between *team familiarity*, complexity and team performance, has been investigated by Espinosa et al. (2007), through research relating to geographically dispersed software development teams. The following two research questions were proposed:

1. "Whether *team familiarity* and *geographical dispersion* have a positive effect on team performance such that the effect of team familiarity on team performance is more evident when teams are geographically dispersed?"
2. "Whether *team familiarity* and *team size* interact positively on team performance such that the effect of team familiarity is more evident when teams are larger?"

What the authors found was that team familiarity helped to mitigate the negative effects associated with team coordination complexity on team performance, relating to both geographically dispersion and team size. It was suggested that team familiarity helps the identification of specific knowledge sources within the team, regardless of location, thus enabling cooperation and responses to any questions to be obtained quicker. With geographically dispersed teams, team members must coordinate their work in some way. It is suggested that such teams do not enjoy the benefit of presence

awareness which could aid the identification of specific knowledge, as well as the benefit of frequent communication and contextual reference.

Previous work, such as carried out by Kelly and McGrath (1985), has emphasised the importance of team interaction to team performance. Subsequent work from Hsu et al. (2011) has endorsed the views of Espinosa et al. by highlighting the importance of the team in sharing and knowledge utilisation, as part of task accomplishment. Contrasting views have been provided by Brooks (1995) who has suggested that larger teams represent an increase in the communication links between team members, which eventually has a negative effect on team performance. In support of Brooks, Espinosa et al. (2007) have concluded that all other things being equal, team performance may indeed decrease when team members are not familiar with each other, but in contrast to Brooks, it is stated that team familiarity not only negates the effects of team size on team performance, but becomes critical as team sizes increases. This is more beneficial in the case of team coordination complexity, whereas other team characteristics such as interaction, coordination, and information sharing are actually challenged. In an endorsement of the research of Espinosa et al (2007), Hsu et al. (2011) have stated the importance of team familiarity, stating that it enables better management and use of information utilisation, in the case of information systems development projects. Team building activities are said to encourage familiarity, and such activities should be directed at improving communications involving all members in problem solving, role clarification, and goal establishment. Activities such as team building are stated are being especially important for teams with high employee turnover rates. Adopting a more general perspective regarding knowledge within teams, Rus et al. (2001) and Chau and Maurer (2004) have emphasised the importance of "Knowing who knows what". This has been referred to as *directory structure* by Chau and Maurer (2004).

Knowledge Utilisation

Hsu et al. (2011) has focussed on the importance of primary influences such as the availability and acquisition of information within teams, to overcome issues such as project complexity. The complexity and often unstructured nature of Information Systems development projects is acknowledged. Team mental models are described as

being an important aspect in facilitating information utilisation, which in turn helps deal with such issues, helping to improve project performance. The proposed model is based on the input-process-output model, as put forward by McGrath (1966), whereby collective information is shared via interaction to achieve desired outcomes. It was found that continual team-building activities relating to communication, problem solving, goal setting, and role clarification, lead to higher levels of teamwork.

Hsu et al. (2011) also highlight the importance of mental models, as well as the following points relating to knowledge availability and use for successful IS projects:

1. *Understanding team mental models* for Information Systems projects. This is of particular importance when a co-working philosophy must be developed in a short period of time, and time pressures exist regarding developing a working relationship and project goals.
2. *Management interventions and practices*, involving all members in the decision making process, might also facilitate team mental models.
3. *Information utilisation by the team* is affected by the level of common understanding among team members on how to interact with other team members to enable the acquisition of necessary information. Therefore interpersonal skills and communication skills also become relevant.

This section has covered task complexity and the important role which task and team familiarity, and mental models plays in relation to task complexity. Examples of actions towards the reduction of task complexity are provided by Bhattacharya et al. (2010) and de Silva and Balasubramaniam (2012). Bhattacharya et al. (2010) have provided an example of a model proposed to aid the improvement of software verification and validation, through the identification of which software components to debug, test, or refactor first. This model also provides some assistance in defect count prediction of modified code. An example of dealing with complexity associated with the system under test is proposed by de Silva and Balasubramaniam (2012). In that particular case the authors have highlighted the benefit of an automated execution environment in dealing with complexity associated with evolving systems. It was suggested that this aids the easy validation and testing of both structural and behavioural aspects of the software system, helping to deal with increasing

complexity. A broader discussion on the role of knowledge in software development is carried out in the following section.

3.2 The Role of Knowledge in Software Development

Rus et. al. (2001), Leidner et al. (2008), and Nonaka and Von Krogh (2009) have all referred to the important role which knowledge transfer plays in an organisation. The key role which knowledge plays in the software development process has also been stated ((Neisser, 1976), (Wagner & Sternberg, 1985), (Chau & Maurer, 2004), (Ryan & O'Connor, 2009), (Pee, Kankanhalli, & Kim, 2010), (Desai & Shah, 2011), (Grambow, Oberhauser, & Reichert, 2015)), and the importance of providing access to such knowledge ((Chau & Maurer, 2004), (Pee, Kankanhalli, & Kim, 2010), (Rabelo, et al., 2015)). One possible explanation for the importance of knowledge is that it is unlikely that all members of a software team will possess all of the knowledge required for all software development activities, thus activities such as knowledge sharing become important aspects of software development, facilitating the transfer of knowledge between team members (Chau & Maurer, 2004). Knowledge can take the form of being documented or undocumented (Rus.I, Lindvall.M, & Sinha, 2001), is tied to the beliefs of the holder, and is organised by the flow of information, (Nonaka & Takeuchi, 1995). Many authors have made the distinction between two primary types of organisational knowledge, *explicit knowledge* and *tacit knowledge* ((Nonaka & Takeuchi, 1995), (Rus.I, Lindvall.M, & Sinha, 2001), (Zack, McKeen, & Singh, 2009), (Holste & Fields, 2010)). Joia and Lemos (2010) define these knowledge types as:

1. *Explicit knowledge* is described as knowledge which can be codified and transferred easily.
2. *Tacit knowledge* is described as difficult to articulate in writing and is normally acquired through personal experience.

Important to this research is the concept, characteristics, and the role of explicit and tacit knowledge within organisations, both on a conceptual, and a practical basis (

(Polanyi, 1966), (Hansen, Nohria, & Tierney, 1999), (Nonaka & Takeuchi, 1995), (Tsoukas, 2002), (Nonaka & Von Krogh, 2009)). There are acknowledged benefits associated with explicit knowledge, such as reducing organisational uncertainty, facilitated through the easy transfer of knowledge, using mediums such as periodical reports, rules, operational standards, procedures and data analysis (Daft, Lengel, & Trevino, 1987).

The second type of knowledge, tacit knowledge, is described as difficult to express in formal language, comes from experience, perceptions and values, and is related to context (Joia & Lemos, 2010). It is linked to practical intelligence, along with formal knowledge and general aptitude ((Wagner & Sternberg, 1985)). Not all authors are in full agreement regarding the definition of tacit knowledge. Gottfredson (2002) has disagreed with the clear distinction made between academic intelligence and tacit knowledge, as made by Wagner and Sternberg (1985) and Sternberg et al. (2000). However a concession is made with an acknowledgement that the concept of tacit knowledge does indeed lend itself to a form of wisdom (knowledge), which is generally developed through experience or observation. Tacit knowledge has been described as having the following characteristics according to Wagner and Sternberg (1985):

- *Practical rather than academic.*
- *Informal rather than formal.*
- *Tacit rather than directly taught.*

Polanyi (1966) has considered tacit knowledge to be something personal, an ability or skill, enabling one to do something or solve a problem, which is partly based on one's own experience and learning. As long as one uses appropriate language, a good deal of knowledge is described as knowledge which can be shared easily among people. Chau and Maurer (2004) described tacit knowledge as knowledge which is not usually documented, and does not tend to be explicitly taught through formal training. To facilitate knowledge transfer, in the case of tacit knowledge, there is a particular dependence on individuals to engage in the practise of knowledge sharing ((Hansen, Nohria, & Tierney, 1999), (Chau & Maurer, 2004), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007), (Pee, Kankanhalli, & Kim, 2010), (Desai & Shah, 2011)). The

challenges associated with the effort and willingness of team members to facilitate the transfer of tacit knowledge has been highlighted by Pee et al. (2010) and Desai and Shah (2011). The relevance of knowledge transfer to tacit knowledge is discussed in greater detail in a forthcoming section. A subsequent section discusses knowledge transfer and knowledge sharing, an area which has been identified as important to software development ((Chau & Maurer, 2004), (Pee, Kankanhalli, & Kim, 2010), (Cataldo & Ehrlich, 2012)), but which regularly faces significant challenges ((Pee, Kankanhalli, & Kim, 2010), (Desai & Shah, 2011)).

Desai and Shah (2011) have highlighted the strong link between knowledge management and software testing, stating that effective management of such knowledge is essential to improving the quality of software testing. The approach to knowledge management is something which has been shown to have a significant effect on the role of tacit knowledge within organisations ((Hansen, Nohria, & Tierney, 1999), (Leidner, Alavi, & Kayworth, 2008), (Kimble, 2013)). In keeping with the second goal of this research i.e. *the relationship between system test complexity and tacit knowledge*, there is a significant focus on the role which tacit knowledge plays in software development environments, and in particular, the role which it plays in software development tasks such as system testing. The importance of both explicit knowledge and tacit knowledge has been emphasised by numerous authors ((Chau, Maurer, & Melnik, 2003), (Desai & Shah, 2011), (Cataldo & Ehrlich, 2012)). The case for research in the area of tacit knowledge (Joia & Lemos, 2010), and an emphasis of the need for a greater understanding of this particular topic and the role which it plays in software development processes, has been made by Ryan and O'Connor (2009) and Dingsøyr and Šmite (2014).

3.2.1 The Importance of Knowledge Sharing

Nonaka and Von Krogh (2009) have emphasised knowledge transfer as a critical component of the learning process, enabling the sharing of employee's experiences, mental models, and their beliefs and perspectives, so that knowledge is made available to others. The combination of knowledge received from other sources, with one's own insights and beliefs, is described as contributing to the creation of new knowledge. The

benefits of knowledge sharing in terms of creativity have also been highlighted by Wang et al. (2012). Knowledge sharing can be ad-hoc or organised within a project or organisation, facilitated through formal communication. Dorairaj et al. (2012) and Wang et al. (2012) have highlighted the importance of knowledge sharing to the software development process, providing a clear link between the role of knowledge and the success of software development teams. Chau and Maurer (2004) suggested that it is most likely that there will always be some dependence on the knowledge of colleagues amongst software development teams, and that it is unlikely that every team members will possess all of the required knowledge to carry out all software development activities.

The importance of the role of knowledge sharing is emphasised in the case of geographically distributed work teams, an increasingly common characteristic of software development environments ((Rus.I, Lindvall.M, & Sinha, 2001), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007)). Groups need to communicate and collaborate, irrespective of time and location, and knowledge sharing is an important element of such work arrangements, facilitating collaboration. The impact of knowledge on the performance of a geographical dispersed team has been stated by Espinosa et al. (2007). Knowledge is stated as playing a critical role, as the size of a geographically dispersed team increases. Both knowledge relating to task familiarity, and directory structure (knowing where to locate specific knowledge within the team), are important elements of successful team performance (similar points have been echoed by Chau and Maurer (2004)). The aforementioned factors are said have a substitutive rather than a complementary relationship, as either type of knowledge increases. An explanation for this is that having more task knowledge makes one less dependent on colleagues, whereas having knowledge as to who holds what expertise, makes one less dependent on task expertise (Espinosa, Slaughter, Kraut, & Herbsleb, 2007). The importance of knowledge to the system test process has been emphasised by Eickelmann and Richardson (1996), and Desai and Shah (2011).

Chau et al. (2003), Turk et al. (2005), and Moe et al. (2012), have acknowledged the relationship between the applied development methodology, the approach to knowledge management, and knowledge sharing. Some software development methodologies such as agile have been described as being heavily reliant on the

communication of tacit knowledge via interpersonal contact. Chau at al. (2003) have referred to traditional software development as striving to achieve idealistic goals via Tayloristic processes. Such traditional models are described as relying on explicit documentation in order to provide the process and product information, to enable team members to effectively achieve their goals (Turk, France, & Rumpe, 2005). Handovers between stages are primarily document based, incomplete, and often lead to information loss between one development stage and the next (Chau, Maurer, & Melnik, 2003). From another perspective, Traditional, Tayloristic, or Plan-driven methods, are stated as reducing the risk of knowledge loss by investing in lifecycle architectures and plans. This also provides the benefit of enabling the adoption of a definitive stand that when requirements changes are introduced unexpectedly during a project. The downside of this is that one can expect a higher probability of schedule and cost overruns, as a result of adopting such an inflexible approach (Rajagopalan, 2014).

Turk et al (2005) have argued that there is an increased importance of tacit communication via personal contact, given the movement away from traditional development strategies, which many see as rigid, plan driven models (Chau, Maurer, & Melnik, 2003). This has resulted in a decreased reliance on explicit knowledge, through a reduction of the length of communication chains, and a corresponding increased reliance on direct, face-to-face communication, for relevant tacit knowledge. The success of agile development methodologies is based on team members understanding, experience, and their ability and willingness to share applicable, tacit knowledge. This is carried out on a continuous, informal basis, between software development team members, and customers (Turk, France, & Rumpe, 2005). Turk et al. state, that when the team's tacit knowledge is sufficient for the application's life-cycle needs, things work fine, but that there is also the risk that the team will become overly dependent on experts, and may suffer from "corporate memory loss", either of which could result in unrecognized shortfalls in available tacit knowledge. The core characteristics of knowledge sharing are discussed in the following section before a more detailed discussion on the concept of tacit knowledge.

3.2.2 The Core Characteristics of Knowledge Sharing

Pee et al. (2010) have identified the core elements associated with knowledge sharing based on the *communication perspective* of Berlo (1960). The *communication perspective* identifies sender, receiver, channel, transmission, and effect as the basic elements of communication, described as inherent in knowledge sharing.

1. *Sender* relates to the knowledge source.
2. *Receiver* is described as the entity acquiring the knowledge.
3. *Channel* corresponds to the medium through which knowledge is shared. Examples of face to face meetings, computer, phone, documentation etc. are provided.
4. *Transmission* relates to the actual process and activity of sending and receiving knowledge through particular channels. The effectiveness of transmission is impacted by factors such as motivation and the social relationships.
5. *Effect* refers to the end result of any knowledge sharing exercise such as performance, learning, and satisfaction.

Relevant factors which are stated as influencing the source of knowledge are the sources command of language, the ability to express knowledge clearly, experience, credibility, etc. The knowledge recipients ability to utilise knowledge (also referred to by Hsu et al. (2011)) is also described as important along with the richness of the communication channel, the environment in which the communication take place, and the nature of relationships between relevant stakeholders.

In their related investigation of the interdependence of subgroups involved in software development, Pee et al. (2010) have acknowledged the relevance of the theory of social interdependence (credited to Deutsch (1949), but having its origins Lewin (1935)). In line with this theory, Pee at al. (2010) have focussed on the interdependence of *goals*, *tasks* and *rewards* between subgroups, and the influence of goals, tasks and rewards on the immediate and future outcomes of other subgroups. In the context of information systems development, social interdependence is described as playing an important role in understanding knowledge sharing in development projects. Perceived social interdependence is focussed on, rather than actual

interdependence, because in line with the views of Johnson and Johnson (2005), behaviour is determined by how a situation is perceived, rather than objectively assessed. Pee et al. (2010) have identified goal, task and reward interdependence as:

- *Goal interdependence* is described as going beyond goal alignment, and requiring that subgroups goals are not only compatible, but also that there is a perception of a reliance on common goal attainment between subgroups. The goal of successful system completion has been identified as a common goal amongst groups involved in the implementation of information systems. It is stated, in line with the social interdependence theory as outlined by Deutsch (1949), that interactions will be promoted when there is a perceived interdependence between subgroups.

- *Task interdependence* refers to the perception of the extent to which any particular subgroup is dependent on another particular subgroup to successfully carry out their work. When subgroups tasks are perceived as to be interdependent, there is an increased likelihood of the promotion of interactions between subgroups.

- *Reward interdependence* is related to the perception that the rewards of a subgroup are dependent on the performance of another subgroup. Reward interdependence is based on the assignment of rewards to a subgroup and the subsequent effect, if any, on the performance of another subgroup.

As a result of the research by Pee et al., it was found that goal, task, and reward interdependencies are significantly related to the process of knowledge sharing between subgroups which are involved in software development. A strong relationship was found between knowledge sharing, the goal, task and reward interdependencies, and software development project performance. It was also found through this research that perceived goal interdependence, significantly influenced task interdependence. Knowledge sharing was not found to be significantly affected by indirect factors such as prior collaboration history, project phase, team size, project complexity, and project contract type.

Regarding facilitating knowledge sharing, an important consideration in any development environment is a strong relationship between the quality of social interaction ((Ryan & O'Connor, 2009), (Talby, Karen, Hazzan, & Dubinsky, 2006), (Moe, A.B., & Dybå, 2012)). This is discussed in more detail in the following section.

Socialisation Difficulties associated with Knowledge Sharing

Hsu et al. (2011) have highlighted the importance of the work environment in enabling knowledge sharing within teams, along with a required ability to utilise such knowledge. Ryan and O'Connor (2009) have specifically made reference to the important link between tacit knowledge, social interaction, and the achievement of project goals. Talby (2006) and Moe et al. (2012) have stated that the link between social interaction and the achievement of project goals is of particular importance in relation to agile software development. Difficulties associated with knowledge sharing and system testing have been identified by Desai and Shah (2011), who state that a socialisation approach to knowledge sharing, involving the transfer of tacit knowledge between individuals, is described as having certain difficulties, and is affected by the following factors:

1. *General lack of time to identify colleagues in need of specific knowledge.*
2. *Apprehension or fear that sharing may affect job security.*
3. *Low awareness and realization of value and benefit of possessed knowledge to others.*
4. *Dominance in sharing explicit over tacit knowledge such as know-how and experience that requires hand-on learning, observation, dialogue and interactive problem solving.*
5. *Use of strong hierarchy, position-based status and formal power.*
6. *Insufficient capture, evaluation, feedback, communication and tolerance of past mistakes that would enhance individual and organizational learning effects.*
7. *Differences in experience and educational levels.*
8. *Poor verbal/written communication and interpersonal skills.*
9. *Age and gender differences.*
10. *Lack of social network.*

11. *Taking ownership of intellectual property due to fear of not receiving just recognition and accreditation from managers and colleagues.*
12. *Lack of trust in people because they may misuse knowledge or take unjust credit for it.*
13. *Differences in national culture or ethnic background and values and beliefs associated with it.*

Joia and Lemos (2010) have highlighted similar concerns to the ones highlighted above, but also, in line with the core characteristics identified by Pee et al. (2010), they have highlighted *transmission* and *communication channel* impacts, detailing factors such as *time management issues, common language, mutual trust, relationship network, type of training, knowledge transference (*is the organisational capable of explicit knowledge management?*),knowledge storage*, *power*, *favourable environment for questioning, type of valued knowledge* (whether it's embodied tacit knowledge), and *media used.* As well as possible difficulties associated with the transfer of knowledge, also highlighted are incentives in the form of rewards. Rewards are core to knowledge sharing ((Rus.I, Lindvall.M, & Sinha, 2001), (Joia & Lemos, 2010)), and should form part of employees' goals, covering both those with considerable expertise, and those that facilitate the transfer of knowledge. Rewards should also cover both know-how as well as formal knowledge. The introduction of penalties to encourage the transfer of tacit knowledge is not considered a viable alternative (Joia & Lemos, 2010).

This section has dealt with the role of both explicit and tacit knowledge in software development. The following section discusses the specific concept of tacit knowledge in greater detail.

3.3 Detailed Discussion on Explicit and Tacit Knowledge

Previous sections have highlighted the role which both explicit knowledge and tacit knowledge plays in Traditional and Agile software development environments. In an Agile development environment, that there is a greater potential for formal documentation and explicit knowledge, to be replaced by informal communications

among software development team members, via continuous feedback between development teams and customers (Turk, al., France, & Rumpe, 2000). The following sections highlight the different perspectives relating to the concept of explicit and tacit knowledge, a term credited to Polanyi (1966). Discussed are contrasting views of authors such as Hansel et al. (1999), Nonaka and Von Krogh (2009), Tsoukas (2003), Ribeiro and Collins (2007), with reference to the *concept of tacit knowledge*, *knowledge creation*, and *the theory of knowledge conversion*. The increasingly important role which tacit knowledge plays in the software development process has been emphasised by Rus et al. (2001), and the necessity for a greater understanding of this particular topic, has been expressed by Ryan and O'Connor (2009) and Dingsøyr and Šmite (2014).

3.3.1 Explicit Knowledge/Tacit Knowledge debate

Whereas explicit knowledge is stated as having universal character, employed consciously, and not tied to any particular context. Tacit knowledge is described as being tied to actions, procedures, commitments, ideals, values and emotions, with a strong relationship to past experiences, true beliefs, and the actions of intuition, and implicit rules of thumb (Nonaka & Von Krogh, 2009). Holste and Fields (2010) have described tacit knowledge in similar terms, as being tied to ones abilities, developed skills, experiences, undocumented processes, and ''gut-feelings'', etc. It is not surprising that the concept such as intuition, described as where one is unable to consciously account for the relationship, between problem, and solution (Dane & Pratt, 2007), is identified as having a strong relationship to tacit knowledge (Nonaka & Von Krogh, 2009).

Tacit knowledge is described as being acquired with little environmental support, and not through formal means (Ryan & O'Connor, 2009). Nonaka and Von Krogh (2009) have asked a number of questions relating to organisational knowledge creation, and the relationship between explicit knowledge and tacit knowledge. Explicit knowledge and tacit knowledge are described as both being conceptually distinguishable along a continuum. Tacit knowledge is described as being accessible through consciousness if it leans towards the explicit side of the continuum. However, most of the knowledge

relating to skills, due to their embodiment, is described as being inaccessible through consciousness. The view of Nonaka and Von Krogh (2009) differ from the views of Polanyi (1966) and Tsoukas (2003) regarding the proposition of *the concept of knowledge externalisation* (conversion of tacit knowledge to explicit knowledge).

As previously highlighted, Polanyi (1966) has considered tacit knowledge to be something personal, an ability or skill to do something or solve a problem, partly based on one's own experience and learning. As long as one uses appropriate language, a good deal of knowledge can be shared among people but not all knowledge. Numerous authors have referred to the benefits associated with attempting to make knowledge within an organisation explicit and available ((Basili, Lindvall, & Costa, 2001), (Hansen, Nohria, & Tierney, 1999), (Ryan & O'Connor, 2009)). These views are based on the assumption that a significant amount of knowledge within organisations can actually be made available as explicit knowledge, and therefore can be stored in knowledge and experience management databases. Ryan and O'Connor (2009) maintained that some tacit knowledge can be articulated, and can therefore be transformed into explicit knowledge, which may be useful for team performance within organisations.

Acknowledging the concept of tacit knowledge to explicit knowledge conversion, authors such as Hansen et al. (1999) have warned against the dangers of attempting to convert the majority of knowledge within an organisation to explicit knowledge, citing a spectacular failure at Xerox. Xerox attempted to replicate the expertise of service-men into an expert system, embedded in their photocopiers. It eventually transpired that the expert system could not replicate the knowledge which is necessary to deal with every different issue which was resolved by the service and repair men on a regular basis, some of which was on the job knowledge acquired on a regular basis, through sharing knowledge between the employees.

The acquisition of tacit knowledge, such as that employed by the Xerox servicemen in order to solve field issues (Hansen, Nohria, & Tierney, 1999), is something which has been touched on by Steinberg et al. (2000), and Ryan and O'Connor (2009). Research by Ryan and O'Connor has found that tacit knowledge affecting team performance on successful software projects is not actually written down, and formalised in work

practices, rather it's more practical or work experienced based. This view is an endorsement of the similarly held views of Polanyi (1966) and Nonaka and Takeuchi (1995). As previously mentioned, it has been suggested that some tacit knowledge can be articulated, and can therefore be transformed into explicit knowledge which may be useful to team performance within organisations, from a general software perspective ((Ryan & O'Connor, 2009), (Holste & Fields, 2010), (Joia & Lemos, 2010)), and specifically from a geographically distributed development team perspective (Espinosa, Slaughter, Kraut, & Herbsleb, 2007). This conversion of knowledge has been dealt with in detail as part of Nonaka's *knowledge creation theory* (Nonaka, 1994), and *knowledge conversion theory* (Nonaka & Takeuchi, 1995), both of which are discussed in the forthcoming section.

3.3.2 Knowledge Conversion

Daft et al. (1987), Hansen et al (1999), Leidner et al. (2008), Nonaka and Krogh (2009), and Murphy and Salamone (2013) have all highlighted the importance of making created knowledge widely available, and connected to an organization's knowledge system. An area which has also been discussed in great details by authors is the conversion of knowledge from tacit knowledge to explicit knowledge ((Nonaka & Takeuchi, 1995), (Hansen, Nohria, & Tierney, 1999), (Tsoukas, 2002), (Ribeiro, 2007), (Murphy & Salomone, 2013)). Notwithstanding the importance of groups to organisations, Rus et al. (2001) have stated that ultimately it is the individual who performs tasks in any attempt to achieve organisational goals, and therefore within any organisation, knowledge and learning at the individual level is of the utmost importance. The work of groups is described as being wholly dependent on the ability of the individual group members, to apply their knowledge ((Rus.I, Lindvall.M, & Sinha, 2001), (Tsoukas, 2002)). Knowledge conversion, something which happens at the individual level, is something which Nonaka and Von Krogh (2009) believe helps explain the interaction between explicit knowledge and tacit knowledge. To give credibility to the argument of knowledge conversion, Nonaka and Takeuchi (1995) use an example of *Matshusita's bread-making machine*:

A product development group at a company was failing to produce a product that could produce good bread. The issue they had is described as being technical; the main issue was that the dough could not be kneaded in a way that brought sufficient air and lightness to the bread. A young engineer named Tanaka acquired the necessary tacit knowledge required to adequately knead the dough from jointly working with a local master baker at a nearby hotel. Upon returning to the company Tanaka made the knowledge explicit by illustrating to the product development group how the master baker handled and kneaded the dough.

Analysis of the story has been provided by Ribeiro and Collins (2007). In a bid to clarify the distinction between tacit knowledge and explicit knowledge, and building on further work by Collins and Kusch (1998), Ribeiro and Collins have distinguished between *polimorphic behaviour*, which is used in describing the tacit knowledge element of the master bread maker, and *mimeomorphic behaviour,* referred to as a mimicking of original behaviour of the baker, which was based primarily on tacit knowledge. Kneading, although it proved to be a process which could be imitated, is described as mastered only as a piece of tacit knowledge by humans, described in similar terms to riding a bicycle. The authors argue that imitation of behaviours based primarily on tacit knowledge i.e. mimeomorphic behaviour, does not necessarily equate to similar behaviour in related circumstances, which the original tacit knowledge, would enable. Riding a bike is given as an example; because you can automate the balance associated with riding a bike, which could be determined as mimeomorphic behaviour, this does not mean that you necessarily appreciate all the nuances with riding a bike, such as riding a bike in traffic etc. something which could be considered polimorphic behaviour. The way the bread-maker mixes and kneads, differs from the way it is done by the Japanese bread-making machine and probably differs from the way humans do it, but in this case the imitation of the exact behaviour associated with the kneading act, proved adequate for machine performance.

Ribeiro and Collins (2007) have provided support for Nonaka and Takeuchi (1995) as part of the distinction made between explicit knowledge and tacit knowledge. However, it is argued that at the end of the example given by Nonaka and Takeuchi (1995), that the master baker's tacit knowledge has been neither fully explained, nor incorporated into the bread-making machine. Advice and instructions may aid the

mastery of polimorphic actions, but the advice cannot replace experience associated with such actions (Hedesstrom, 2000).

Tsoukas (2002) has argued that tacit knowledge conversion is not sustainable. This argument, in line with the views of Polanyi (1962), is that tacit knowledge and explicit knowledge are not two ends of a continuum but rather described as "two sides of the same coin", with even the most explicit of knowledge is supported by tacit knowledge. Tsoukas accepts the views expressed by Polanyi (1962), that tacit knowledge consists as a set of supporting ancillary knowledge, which we are aware of as we focus on something else. An example is provided related to the task of hammering a nail. The primary focus is on the nail, with tacit knowledge manifesting itself as effable knowledge of either the hammer or the swing. According to Tsoukas (2002), tacit knowledge is completely intertwined with the associated focus with which it is linked and efforts to separate tacit knowledge from that focus, for it to be examined independently, risks losing the true meaning of such knowledge. Thus the true meaning of such knowledge cannot be articulated, and is therefore lost in conversion.

It is argued that the meaning of tacit knowledge is derived from the connection to a particular focus. When we focus on a new set of particulars, it is a new context of action, demanding a new set of ancillary knowledge, thus rendering the notion of a conversion from tacit knowledge to explicit knowledge, as outlined by Nonaka and Takeuchi (1995), unsustainable. Even though, in line with the aforementioned views, it is maintained that we cannot fully discuss skilled performances in which we are involved, Tsoukas does entertain the notion that we can command a clearer view a particular tasks, if we remind ourselves of how we do things. If done, distinctions which we had not previously noticed may be brought to our immediate attention. Contrary to the views of Ambrosini and Bowman (2001), which enforces the necessity to externalise (make explicit) tacit knowledge, it is argued that we need to find new methods of talking, connecting and interacting, in order to create tacit knowledge. Tacit knowledge cannot be captured, translated, or converted, only displayed in what we do. Therefore it is only through social interaction that new knowledge is created (Tsoukas, 2002).

Nonaka and Von Krogh (2009) have stated that there are three major aspects to arguments against the notion of knowledge conversion from explicit knowledge to tacit knowledge:

1. *The conceptual basis.*
2. *The relationship of knowledge conversion in social practice.*
3. *The outcome of knowledge conversion.*

Cases against knowledge conversion on the conceptual basis are described as being based on the original views of Polanyi (1966). The accepted premise is that tacit is knowledge is essentially inexpressible, therefore it can never be converted or externalised and written down in explicit form. Another argument stream centres around the relationship between knowledge conversion and social practice, based on the view that in Nonaka and Takeuchi (1995) "Matshusita's bread-making machine" story, that Tanaka acquired tacit knowledge by working jointly with the master baker. This view that tacit knowledge is only ever created through social interaction has been emphasised by Hedesstrom (2000), and Tsoukas (2002). The third argument stream, relates to the outcome of knowledge conversion. As previously referred to, author's such as Ribeiro and Collins (2007) have argued that although certain aspects of the master baker's behaviour was incorporated into the bread making machine, that this merely relates to an imitation of certain aspects of the master baker's bread making process, and does not constitute a conversion to explicit knowledge, of any aspect of the master baker's tacit knowledge.

Nonaka and Von Krogh (2009) responded to the criticism of both Tsoukas (2002) and Ribeiro and Collins (2007), as part of an attempt to justify the concept of knowledge conversion in the face of criticism. Nonaka and Von Krogh justified knowledge conversion, based on the premise of a tacit knowledge to explicit knowledge continuum, figure 3.4.

Tacit Knowledge to Explicit Knowledge Continuum (Nonaka and Takeuchi (1995))

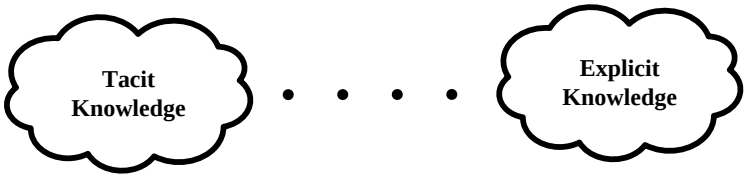


Figure 3.4: Tacit Knowledge to Explicit Knowledge Continuum..

The premise of all explicit knowledge being founded in tacit knowledge is core to the argument of Nonaka and Von Krogh (2009). They have referred to the views of Day (2005), who has stated that some tacit knowledge must be the basis for explicit knowledge. The work of scientists is given as an example. The centre of all scientific investigation must be the ability to make explicit, tacit knowledge relating to such things as discovery processes, the results of scientific improvisations with instruments in the laboratory, and errors to avoid when attempting replicate the experiments. Thus, it has been suggested by Day, and by Nonaka and Von Krogh, that some knowledge may move along the continuum, from tacit knowledge to explicit scientific knowledge, to become knowledge which is independent of the scientist themselves. Hedesstrom (2000) has made an attempt at reconciling the views of Nonaka and Von Krogh, Polanyi (1966), and Tsoukas (2002). They state that the views of the aforementioned authors can be encapsulated by distinguishing between tacit knowledge: which has not yet been formalised because of the following reasons:

1. *Tacit knowledge which has not been formalised because of cost or time limitations.*
2. *Tacit knowledge which has not been formalised because of the form of the knowledge, such as embodied knowledge.*

The concept of an explicit knowledge to tacit knowledge continuum is increasingly being discussed in literature (Hedesstrom, 2000). Such a continuum backbones the argument by Nonaka and Von Krogh (2009) relating to knowledge conversion, also enabling organisations the capability of distinguishing between organisational knowledge assets which are quite tangible, such as technology and procedures, and

knowledge which is described as demanding "thick" levels of interpretation, such as organisation culture or expertise. The ability to make such distinctions between the knowledge assets of a firm is said to aid management practice. The authors clarify their viewpoint, stating in line with the organisational knowledge creation theory, that not all knowledge is capable of being made explicit. Knowledge relating to physiology, sensory and motor function, is stated as not lending itself to being articulated and detailed. The argument which Ribeiro and Collins (2007) have made, regarding the master bakers tacit knowledge being explicated and made explicit is described as a misinterpretation of the original text of Nonaka and Takeuchi (1995). The argument is made that some explicit knowledge can enable machines to solve very specific, constrained problems, but as referred to by Dreyfus and Dreyfus (1986), expert knowledge can never be fully captured in computer software due to the existence of embodied tacit knowledge.

In support of the theory of knowledge conversion, Nonaka and Von Krogh (2009) have stated that after individuals acquire explicit knowledge, that knowledge is internalised through acting on that acquired knowledge, through action, practise and reflection. To reinforce this view, Nonaka and Von Krogh, have detailed studies, such as those carried by Chou and He (2004), as providing evidence of knowledge conversion. A survey was conducted of 204 organisations in a variety of industries, with a concentration on knowledge conversion and knowledge assets. As part of this research a distinction was made between four different types of organisational knowledge assets:

1. *Experiential knowledge assets:* this type of tacit knowledge asset is built through shared hands-on experience amongst members of an organisation. It is stated as also relating to emotional knowledge such as care, love, and trust.
2. *Conceptual knowledge assets:* this knowledge is described as explicit knowledge, articulated through images, symbols, and language. Such assets are described as communicated through models, analogies, and metaphors.
3. *Systemic knowledge assets:* this consists of systematic and packaged explicit knowledge, consisting of elements such as technologies, product specifications, manuals, and organisational documents and information. Such knowledge is often stored in a knowledge repository.

4. *Routine knowledge assets:* this consists of tacit knowledge relating to the routine work practices and actions of an organisation.

The relationship between these knowledge assets and the following knowledge conversion variables was investigated:

- *Socialisation*: the process of creating tacit knowledge through shared experience.
- *Internalisation*: the process of embodying explicit knowledge into tacit knowledge.
- *Externalisation*: the process of embodying tacit knowledge into explicit knowledge.
- *Combination*: the combination, editing and processing of explicit knowledge to form new explicit knowledge.

Knowledge Assets and Knowledge Conversion (Chou and He (2004))

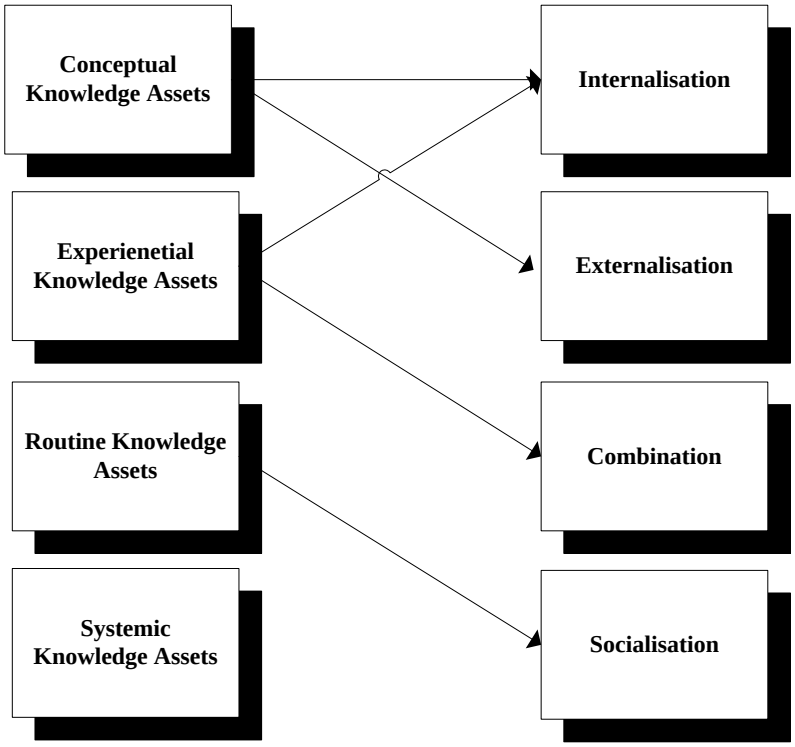


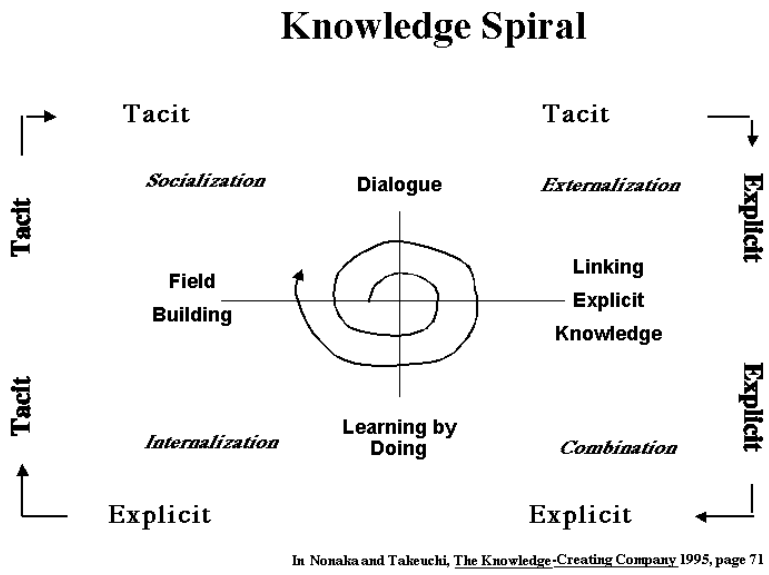
Figure 3.5: Knowledge Assets and Knowledge Conversion.

Figure 3.5 details the relationships that were found as a result of the study carried out by Chou and He (2004). It was found that all knowledge assets contributed to knowledge creation, with conceptual knowledge having the most significant effect, and systemic knowledge assets found to have the least significant effect on knowledge creation. Internalization and externalization variables were found to have the strongest relationship to conceptual knowledge assets. Experiential knowledge was found to have a strong relationship to the combination variable and internalisation. There was a weak relationship found between systemic knowledge assets and the combination variable but this was not deemed significant enough, in comparison to the other relationships, to be detailed.

In further support of Nonaka and Von Krogh (2009), and the concept of knowledge conversion, are the views expressed by Sun et al. (2001). A model for skill learning, based on implicit (procedural knowledge) is developed. A distinction is made between top-down knowledge, whereby declarative knowledge is turned into procedural knowledge through practise, and bottom-up knowledge where procedural knowledge comes first and then declarative knowledge i.e. implicit knowledge comes first, then explicit knowledge relating to how to perform a particular task. It is claimed that models adopting a top-down approach, are more common in literature, with most claimed to focus on learning, taking instructions/examples and turning them into procedural skills. Investigation is carried out by the authors on a bottom-up approach, allowing for the capture of both procedural and declarative knowledge, with the acquisition of procedural knowledge prior to, or simultaneous to, the acquisition of declarative knowledge. When there is no sufficient, relevant, a priori knowledge available, learning may occur on a bottom-up basis, with implicit knowledge the primary influence, and explicit knowledge the secondary influence. Under such circumstances, on undertaking a task, relevant past experiences are retrieved implicitly, with a response selected. Such responses may be based on previously stored instances, or the summarisation of instances. These instances are used by comparing against a current situation, and depending on similarity, a response is formed which is specific to the current situation.

In line with the views of Anderson (1983), Sun et al. (2001) have stated that, when skill has been derived from declarative knowledge, that over time and with practice, that procedural knowledge can be used with minimal declarative knowledge necessary. The reference to the externalisation of explicit knowledge from tacit knowledge provides a strong link to the work of Nonaka and Von Krogh (2009). It has been proposed that explicit knowledge lags behind tacit knowledge and is actually extracted from tacit knowledge. Although, in line with the model proposed by Sun et al. (2001), declarative knowledge plays a secondary role in bottom-up skill development, the importance of declarative knowledge is emphasised as often speeding up the learning process, facilitating the transfer of skill by speeding up learning in new settings. Declarative knowledge is also stated as facilitating the communication of knowledge.

An example of explicit to tacit knowledge conversion is stated as being provided through the work of Ashby et al. (1998), who carried out research as to the existence of separate verbal and implicit learning systems, deemed particularly important in the case of individuals with learning, verbal, and memory affecting medical conditions, such as Parkinson's and Amnesia. It was found that an individual may acquire explicit knowledge through both verbal and procedural learning systems, and that with training and experience, that tacit knowledge may actually become more important in solving that task over time. Further credibility is given to the idea of knowledge conversion by Rus et al. (2001) who have referred to the relevance of the knowledge transformation spiral, figure 3.6 (as outlined by Nonaka and Takeuchi (1995)), in their development of an approach to knowledge management.



(Nonaka and Takeuchi (1995))

Figure 3.6: The Knowledge Spiral.

According to Rus et al. (2001), a significant principle of the spiral is that that knowledge is enriched when shared and is not diminished through use. In order for knowledge to be transferrable, it must first be transformed into information (externalised) which involves the process of capturing information about knowledge. Knowledge must then be converted back from information into knowledge (internalised), which involves the process of understanding, putting it into context with one's own existing knowledge, thereby transforming the information into knowledge.

Some concerns have been highlighted regarding application of the model as proposed by Nonaka and Takeuchi (1995), which Nonaka and Von Krogh (2009) take the time to address. As previously referred to, Nonaka and Von Krogh (2009) have asked a number of questions relating to organisational knowledge creation and the relationship between explicit knowledge and tacit knowledge. Both explicit knowledge and tacit knowledge are described as both being conceptually distinguishable along a continuum. Tacit knowledge is described as being accessible through consciousness, if it leans towards the explicit side of the continuum. It is suggested that tacit knowledge is bound by rules associated with social practice. An example is given of a pianist, who learns the rules of performance including skills, values, beliefs, and norms associated

with the social practice of piano playing. An argument against a tacit / explicit continuum is provided by Tsoukas (2003), who appear to enforce the opinion of all tacit knowledge being embodied, providing Polanyi's (1962) analysis in relation to map reading. No matter how elaborate a map is, it cannot read itself but rather requires the judgement of a skilled reader who will relate the map to the world through both cognitive and sensual means (Polanyi, 1962).

It is this description of tacit knowledge, as being very much embodied, which is at odds with the views of Nonaka and von Krogh (2009). The identification of aspects of tacit knowledge, such as social considerations, albeit we may be subconsciously aware of them, appears to separate the views of Polanyi (1962), and Nonaka and Von Krogh (2009). Social practices in organisations, involving members with varying experiences of different social practices (and thus diverse tacit knowledge), have been argued by Nonaka and Von Krogh (2009), as being an important source of knowledge creativity. It is argued through knowledge conversion (externalisation and combination), that a member's diverse tacit knowledge, at least partly acquired through diverse social practices, can lead to new ways of defining problems, and new ways of searching for solutions. Knowledge creativity is a topic which has been identified as an important element of software testing (Desai & Shah, 2011).

This section has dealt with the subject of tacit knowledge. Desai and Shah (2011) have highlighted the importance of a structured approach to knowledge management, in the case of software testing. This is described as being of particular importance in the case of tacit knowledge and something which can eventually result in a reduction in time, cost, and effort, for software testing. Knowledge management is discussed in greater detail in the following section.

3.4 Approaches to Knowledge Management

This management of knowledge in software engineering relates to the ultimate goal of capitalizing on an organisations intellectual capital, something which is described as important to the software development process ((Rus.I, Lindvall.M, & Sinha, 2001),

(Dingsøyr & Šmite, 2014)). This goal is achieved through a process of knowledge creation, sharing, and capture in organisations (Nonaka & Von Krogh, 2009). Desai and Shah (2011) identified the benefits of a managed approach to both explicit and tacit knowledge, particularly in the case of software testing. Frameworks have been put forward for the management of knowledge in a software development environment, enabling consistent access to knowledge ((Rus.I, Lindvall.M, & Sinha, 2001), (Von Krogh, 2012)). In a review of empirical studies relating to knowledge management of global software development projects, Dingsøyr & Šmite (2014) have put forward five common approaches to knowledge management:

1. *Systems school*: related to the application of techonlogy for knowledge management e.g. knowledge repositories.
2. *Cartographic school*: related to the knowledge maps and the creation of knowledge directories. Such an approach is useful for storing knowledge relating to resources, skills, projects opportunities etc.
3. *Engineering school*: such an approach focusses on processes and knowledge flow within organisations. This has been referred to as focussing primarily on processes for mapping knowledge, conducting project retrospectives, accomodating mentoring programs, and catering for detail relating to work processes e.g. CMM (the capability maturity model). This model is stated as being primarily based on explcit knowledge.
4. *Organisation school*: this approach is concerned with networks for sharing or pooling knowledge. This is often put into practise by way of *communities of practise* related to a common topic of interest. It is stated that such communities facilitate the transfer of both tacit knowledge and explcit knowledge, with the explcit knowledge transfer, typically being less formal than the case of knowledge repositories.
5. *Spatial school*: this approach is related to how an office space can facilitate the knowledge management. This can range from setting up whiteboards, to the use of an open plan office structure to encourage engagement. A popular use in the case of an agile approach to software development, is the use of taskboads, which relate to project status and visible to stakeholders. This approach is staed as being dependent on colocation and appears to work well for smaller teams.

Those global organisations employing a more traditional approach to software development are stated as predominantly relying on *systems* or *engineering* schools, whereas those working in accordance with agile methodologies are stated as relying on *spatial* and *organisational* schools. The *cartographic* school is stated as providing a cost-effective means of knowledge management, irrespective of the employed development methodology.

Even with the recognised knowledge management systems that are available, providing required knowledge to the appropriate people within organisations, still remains a major issue (Grambow, Oberhauser, & Reichert, 2015). This might be explained by the fact that such approaches demand a considerable time and effort, both at an individual and organisational level (Rus.I, Lindvall.M, & Sinha, 2001). The following sections deal with some concerns associated with the management of knowledge.

3.4.1 Consideration of the Development Environment

The aforementioned might be explained in some part, by the fact that there are different approaches required regarding knowledge management, depending on the software development approach which is being applied. For instance, Chau and Maurer (2004) described Tayloristic and Agile methods as necessitating different training mechanisms, to encourage the transfer of knowledge. Formal training sessions are required in the case of Tayloristic methods, and informal practices in the case of agile methods. An example given is pair programming, used in the case of XP, which involves software developers carrying out work in pairs. Formal training has the advantage of allowing training content and practices, to be standardized and applied consistently, across organizational teams. The downside is that formal training is expensive, resulting in a loss of development time for both the trainers and the trainees. It is claimed that informal training practices, as applied in the case of Agile practices such as XP, can result in learning curves being significantly reduced,

communication and coordination improved, and the sharing of tacit knowledge facilitated.

Rus et al. (2001) have highlighted some of the drivers in software development organisations, for the adoption of knowledge management approaches. These drivers relate to both business needs, and knowledge needs:

Business needs:

- *Decreasing time and costs, and increasing quality*: This primarily relates to an avoidance of mistakes relating to previous projects, through the acknowledgement and explicit documentation of such process knowledge, enabling ease of access for future projects.
- *Enabling better decision making*: Leveraging of individual knowledge to enable better decision making to be made at group, and organisational levels.

Knowledge needs:

- *Acquiring knowledge about new technologies*: organisations must acquire knowledge quickly about newly adopted technologies in order to avoid delays associated with learning *by doing approach.*
- *Accessing domain knowledge*: Software development requires domain knowledge relating to not only the system and development environment, but also relating to the final deployment site.
- *Sharing knowledge regarding local policies and practices*: while the informal dissemination of knowledge relating to software development practices is important, such knowledge should be made formal, where possible. This allows all organisational employees to benefit from access to such knowledge.
- *Capturing knowledge relating to who knows what:* knowledge of "who knows what" within an organisation is essential to creating a strategy. The goal of which is to avoid a situation which may occur through attrition, whereby knowledge is not fully appreciated until it is actually lost. This has been referred to as directory structure by Chau and Maurer (2004), and is described as being primarily tacit in nature. It is stated that people in software organisations spend up to 40% of their work time searching for, and accessing, different types of information related to projects, Henninger (1997). In the

absence of employee's expertise, people are stated as spending 3-4 days of any project locating experts.

- *Collaborating and sharing knowledge:* the collaboration and sharing of knowledge within software development teams is a very important activity, irrespective of the geographical dispersion of the teams.

Basili et al. (2001) have highlighted some similar drivers to that of Rus et al. (2001), specifically those relating to the avoidance of previous project mistakes, employee attrition, organisation processes, and team collaboration i.e.:

- *The costly repetition of mistakes, which if documented from a previous project could have been avoided.*
- *The impact of the sudden departure of an employee.*
- *The lack of knowledge availability regarding current organisational processes or products due to no-documentation of same.*
- *The non-availability of knowledge to enable accurate estimation of potential projects.*

3.4.2 Accommodating an Ad-hoc or Formalised Knowledge Transfer Strategy

Nonaka and Von Krogh (2009) have stated that knowledge transfer can be ad-hoc or organised within a project or organisation, facilitated through communication. If this communication and sharing of knowledge is systematic, and there is a process in place to document it, then exchanged knowledge may be captured and organized into organisational or group memory. Authors such as Leidner (2008) and Hansen et al. (1999) have shown that knowledge sharing is important to all types of organisations, regardless of the knowledge management strategy employed. Leidner et al. (2008) has stated that organisations have traditionally adopted one of two approaches to knowledge management. The first approach involves a focus within the organisation on *communities of practice*, or alternatively, the second approach focuses on *facilitating the process of creation, sharing, and the distribution of knowledge*. While organisations may adopt different aspects of both approaches, both approaches are claimed to present different challenges. The first approach is said to be cognisant of

the fact that a great deal of organisational knowledge is in fact held *tacitly*. Formal processes and technologies are stated as not being suitable for enabling the transmission of such knowledge. The approaches to knowledge management from both a community perspective, and a process perspective, have also been referred to as *personalisation* or *codification* approaches, respectively (Hansen, Nohria, & Tierney, 1999). Facilitating the knowledge of tacit knowledge is of particular importance in the case of a *personalisation*/*communities of practice* approach to knowledge management ((Hansen, Nohria, & Tierney, 1999), (Leidner, Alavi, & Kayworth, 2008)).

Hansen et al. (1999) have provided examples of different approaches to knowledge management and knowledge sharing, as employed by technology giants such as Dell and HP. Dell are described as investing heavily in an *codification* approach, providing access to knowledge using electronic storage and access to knowledge, whereas HP are described as adopting a *personalisation* approach, investing in enabling efficient access to personal (tacit) knowledge, enabled by actively promoting person to person meetings, albeit at significant organisational cost. Another key point raised, is that firms which rely heavily on explicit knowledge, tend to fare better with a codification (externalisation) approach to knowledge sharing, whereas firms which rely predominantly on tacit knowledge tend to fare better with a personalisation (or socialisation focussed) approach to knowledge sharing. These views would appear to be in line with the views of Chou and He (2004). The following sections discusses in more detail, personalisation and codification approaches to knowledge management.

3.4.3 Adoption of a Personalisation Approach to Knowledge Management

Rus et al. (2001) have made reference to *communities of practice* approach to knowledge management, whereby a group of individuals team up to work on a project, or develop a product. Such an approach has also been referred to as a *personalisation* approach by Hansen et al. (1999). This approach to knowledge management recognises social environments and communities, as the primary means for facilitating the sharing of knowledge ((Leidner, Alavi, & Kayworth, 2008), (Von Krogh, 2012)). Other communities and organisational groups which facilitate the exchange of

information in different settings and for different purposes have been referred to by Agresti (2003):

- *Community of practice (COP)*: This includes people performing similar work activities.
- *Community of expertise (COE)*: These individuals possess high levels of knowledge in the same subject area.
- *Community of interest (COI)*: This group includes those who share an interest in a subject area.
- *Community of learning (COL)*: These people self-organize to learn and grow professionally and personally.
- *Project team*: These individuals come together as a group for a specified period of time to do a job and then disband.
- *Task force*: This group has similar attributes of a project team, but in this case people work in a totally dedicated fashion with a single objective, working over shorter periods of time, often under a great deal of pressure. This group is also related to a Community of Purpose.
- *High-performance team*: This group is said to possess attributes more closely associated with a true team than the typical group working on a project. They are said to be a highly effective unit, developed over a period of time which often stays together over successive work assignments, growing in maturity and effectiveness.
- *Organizational unit*: These people share membership in an entity defined as part of the organization's structure.

In line with the views of Agresti (2003), Rus et al. (2001) have also highlighted communities as being essential for learning within organisations, particularly in the case of *communities of interest*, and *communities of practice*. Knowledge acquisition is described as potentially occurring from numerous sources such as organisational projects, inter-company learning (such as software vendors and other software development companies), and from industry wide knowledge such as *communities of experts* (guidelines, standards etc.).

A Codification Approach to Knowledge Management

Hansen et al. (1999), Basili et al. (2001) and Leidner et al. (2008) have all made reference to a codification approach to knowledge management. Basili et al (2001) have stated that an improvement of business processes requires that experience be analysed and synthesized, which in turn requires that it be captured, structured, and made available. A number of steps are mentioned as important for an organisation to perform in order to facilitate a codification strategy:

1. The organisation needs to become *less dependent on its employees* in order to mitigate the effects of knowledge loss due to employee departure.
2. The organisation needs to *unload its experts*. The organisation needs to elicit and store the knowledge of experts in order to make available valuable experience.
3. Third, it needs to *create productive employees sooner*. New employees need much information to become productive, but they might not know what they are looking for. The organization needs to package experience in a form that makes it easy for new employees to get up to speed fast without bugging the experts of the organization.
4. Fourth, it needs to *improve its business processes*. Improvement of business processes requires that experience be analysed and synthesized, which in turn requires that it be captured, structured, and made available. Thus the organization needs to model its business processes and make them available to its employees.

Basili et al. (2001) does not appear to endorse the views of Hansen et al. (1999) and Leidner et al. (2008) regarding the eliciting and storage of expert knowledge. The following section deals with literature related to the appropriateness of a personalisation or codification approach to knowledge management.

A Word of Caution Regarding the Selected Knowledge Management Approach

Rather than all firms unloading their experts, a firm may alternatively, adopt a personalisation strategy with regards to knowledge management ((Hansen, Nohria, &

Tierney, 1999), (Leidner, Alavi, & Kayworth, 2008)). A codification (externalisation approach) has been described by Hansen et al. (1999) as consisting of elaborate methods of codifying, storing and reusing knowledge via electronic form. This approach is stated as being practised by consulting firms such as Andersen Consulting and Ernst and Young. Reusable knowledge objects are extracted from the creator and made independent of that person for future use. This people-to-document approach results in the creation of knowledge objects, which may be searched and accessed for information at subsequent stages, without the cooperation of the original creator. A personalisation approach to knowledge management (based on socialisation) is described as being practised by firms such as McKinsey consulting. Which method is used to manage knowledge is described as being wholly dependent on:

1. *The method by which clients are served.* Some customers may require a highly customized innovative solution whereas other customers may require a highly efficient knowledge management system for efficient access to knowledge in future case.
2. *The economics of the business.* Some organisations are described as having a codification strategy based on the "economics of reuse" whereby once a knowledge object is defined it may be communicated electronically and reused effectively repeatedly and at low cost. Other organisations employing a codification strategy rely on "expert economics", whereby tacit knowledge is the primary knowledge type, and knowledge is transferred via a slower personal contact. Such organisations can be highly effective in delivering customised, innovative solutions for customers which extensive networks of personal experts built up within the organisation.
3. *The employees which are hired.* Organisations employing a codification strategy, such as Andersen Consulting, train graduates to work in developing and working with information systems. Employees are aided by the knowledge repository to help develop different scenarios business processes. Employees of such firms are described as implementers and not inventors. The McKinsey and Bain organisations are provided as examples of organisations which employ a personalisation strategy and employ primarily based on analytical skill and innovative capabilities. In such organisations it is essential that employees are

capable of knowledge sharing via person-to-person contact and thus the recruitment process can be somewhat protracted.

Hansen et al. (1999) have stated that knowledge sharing is important to all types of organisations, regardless of the knowledge management strategy employed, and is important at different organisational levels. This point of the importance of knowledge sharing is echoed by other authors, in the context of software development ((Basili, Lindvall, & Costa, 2001), (Chau & Maurer, 2004), (Joia & Lemos, 2010)), and specifically to the task of system testing (Desai & Shah, 2011). Firms which rely heavily on explicit knowledge, tend to fare better with a codification approach to knowledge sharing, whereas firms which rely predominantly on tacit knowledge, tend to fare better with a personalisation approach to knowledge sharing.

Lam (1997) and Hansen et al. (1999) have questioned the necessity for organisations which make significant attempts to externalise tacit knowledge, with Hansen citing the failure of Xerox in their attempts to replace the tacit knowledge associated with service men with an expert system. Also cited are examples of successful approaches to knowledge management, including those that have a knowledge management policy involving the externalisation of tacit knowledge. It is advised that any approach to knowledge management should be taken on a case by case basis, and that organisational strategy, capability, and goals, should all be taken into consideration in development of any such approach. Alternative approaches, as put forward by Wang et al. (2012), enabling knowledge sharing, are described as being more appropriate in the case to a personalisation approach to knowledge management.

Regardless of the approach to software development, Desai and Shah (2011) identified the necessity to manage knowledge with relation to software testing, and the various stages associated with software testing i.e. *test planning*, *test development*, *test management*, *test execution*, *test fault analysis* and *test measurement*. The particular importance of a knowledge management approach has been highlighted as part of this section. The following section provides concluding notes relating to the overall discussion which has taken place relating to both complexity and knowledge.

3.5 Concluding Notes relating to System test Complexity and the Role of Tacit Knowledge

This chapter has provided an overview of system test complexity and its relationship to tacit knowledge. This started with a discussion of literature relating to complexity which may impact the software development process, and specifically the task of system testing. An important concept at the outset of this chapter relates to the views of Brooks (1995), who states that software complexity can be viewed from two different perspectives:

1. *Complexity inherent in software.*
2. *Complexity associated with the process of software development*.

A distinction has been made between essential complexity and accidental complexity associated with software engineering, with difficulties associated with the nature of software, being described as essentially complex, and difficulties associated with the production of software, being described as accidentally complex (Brooks F. P., 1986). Much in keeping with the views of Brooks, McKeen (1994), also focussing on information system development complexity, has made a distinction between *task complexity* and *system complexity*. Task complexity is stated as originating from a user's environment, and relates to ambiguity and uncertainty associated with the practise of business i.e. relating to activities or issues which the system is attempting to address. System complexity originates in the developers environment, and relates to the ambiguity and uncertainty associated with the practise of system development.

In line with the aforementioned views, a number of key perspectives have been highlighted in this chapter, relating to complexity associated with the system under test, and the complexity associated with the process of software development (inclusive of software testing):

- *Inherent software complexity*: This characteristic of inherent complexity, which may affect the specification, design, development, and testing of software, is something which numerous authors have made reference to, from a general software development perspective ((Mumford, 1983), (Brooks F. , 1995),

(Lehman, 1996), (Lyytinen, Mathiassen, & Ropponen, 1998), (de Silva & Balasubramaniam, 2012)), and specifically from a geographically distributed development team perspective (Espinosa, Slaughter, Kraut, & Herbsleb, 2007). The modification of software is potentially a complex activity ((Brooks F. P., 1986), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007), (Bhattacharya, Iliofotou, Neamtiu, & Faloutsos, 2012)).

- *Software task complexity*: The process of software development has been described as an error-prone, time-consuming, labour intensive activity, which can involve considerable complexity, (Akman, Misra, & Cafer, 2011). Other authors have specifically referred to the complexity associated with the task of software testing, (Yeates, Shields, & Helmy, 1994) (Zheng, Alager, & Ormandjieva, 2008), (Debbarma, Singh, Shrivastava, & Mishra, 2011). Examples of actions towards the reduction of task complexity have been discussed in this chapter. Examples of such actions have been provided by Bhattacharya et al. (2010), and de Silva and Balasubramaniam (2012), and concern models relating to test selection, test measurement, and test automation, respectively.

The socio-technical model was also introduced in this chapter, due to its stated benefits in helping to understand the effect of information systems in organisations ((Lyytinen, Mathiassen, & Ropponen, 1998), (Vidgen & Madsen, 2003), (Herbsleb, 2007), (Sommerville I. , 2007), (Lu, Xiang, & Wang, 2011), (Davis, Challenger, Jayewardene, & Clegg, 2013)). The socio-technical model provided in figure 3.7 provides an indication of some of the views which have been expressed in this chapter. A focus on complexity associated with task execution has been provided by Wood (1986) and Campbell (1988). Espinosa et al (2007) and Hsu et al. (2011) have highlighted other influences on task complexity, such as the influence of software development teams (also referred to by Brooks (1995)), and organisational environmental influences, such as the geographical dispersion of teams.

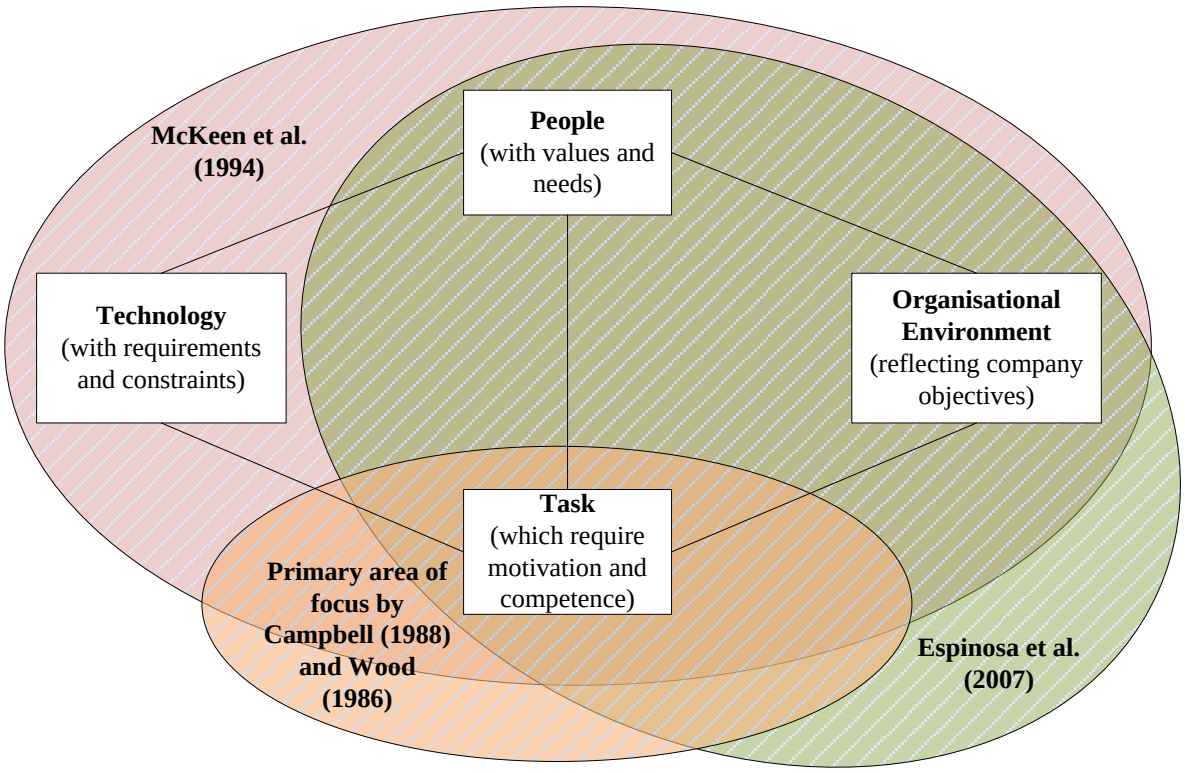


Figure 3.7: Complexity Literature from a Socio-Technical Perspective (based on model by Mumford (1983)).

The strong relationship between complexity associated with aspects of geographically dispersed software development process and knowledge, has been highlighted by Espinosa et al. (2007), Staats et al. (2010), and Wang et al. (2012). Lu et al. (2011) have also acknowledged the complexity of information systems development and have emphasised the necessity of knowledge sharing. This distribution of knowledge amongst team members is particularly important in the case of complex tasks, Staats et al. (2010). The relationship between system test complexity and tacit knowledge is also provided by Staats et al. (2010).

The role of Tacit Knowledge

Nonaka and Von Krogh (2009) have asked a number of questions relating to organisational knowledge creation and the relationship between explicit knowledge

and tacit knowledge. Explicit knowledge and tacit knowledge are described as both being conceptually distinguishable along a continuum, a view acknowledged by Hedesstrom (2000), and supported by Collins and Kusch (1998), and Ribeiro and Collins (2007). Tacit knowledge is described as being accessible through consciousness, if it leans towards the explicit side of the continuum. However, most of the knowledge relating to skills, due to their embodiment, is described as being inaccessible through consciousness. This point has been echoed by Hedesstrom (2000), who have made an attempt at categorising the views of Nonaka and Von Krogh (2009), Polanyi (1966), and Tsoukas (2002). He has stated that the views of the aforementioned authors can be encapsulated by distinguishing between:

- *Tacit knowledge which has not been formalised because of cost or time limitations.*
- *Tacit knowledge which has not been formalised because of the form of the knowledge, such as embodied knowledge.*

Hedesstrom has made reference to the acceptance amongst a growing number of authors, regarding the clear distinction between tacit knowledge and explicit knowledge. In line with the views of Nonaka and Von Krogh (2009), he has referred to the link between some aspects of tacit knowledge (related to actions which are referred as polymorphic in nature) and society. It has been argued that such that actions such as riding a bike through traffic cannot be learned without the consideration of society. Therefore, as society cannot be replicated, advice and instructions may aid the mastery of such polimorphic actions, but the advice cannot replace experience.

This research involves the consideration of knowledge as it applies to the software development process, as discussed by Chau et al. (2003), and Cataldo and Ehrlich (2012). Of particular interest is knowledge as it applies to the task of system testing, as discussed by Desai and Shah (2011). In the previous chapter, knowledge dependency associated with software testing was highlighted as affecting the different stages of system testing i.e. *test planning*, *test development*, *test management*, *test execution*, *test fault analysis* and *test measurement* ((Eickelmann & Richardson, 1996), (Desai & Shah, 2011)). In line with the views of Brooks (1986), and McKeen (1994), two

categories of knowledge are identified below, which have a direct effect on the ability to execute these system test related functions:

1. *Test Knowledge* is an important element in the consideration and achievement of any *test objectives* and *test approaches*. This applies to the task of system testing. This is emphasised by the views of Mattiello-Francisco (2011), which has highlighted the importance of a structured approach to testing, stating that ad-hoc testing is no longer acceptable as an efficient and effective form of testing.

2. *System knowledge* and *knowledge of system requirements*: is a critical aspect of software testing, with some test approaches demanding in depth knowledge of both the system and system requirements. Such a test approach is arguably of greater importance in the case of white box testing, whereby detailed system component testing is being performed, Horgan and Mathur (1996), Lin et al. (2012), and Yoo and Harman (2010).

This chapter has discussed the important role which software testing plays as part of the software development process, and also the significant role which both complexity and tacit knowledge play in this process. The importance of both explicit knowledge and tacit knowledge has been emphasised by numerous authors ((Chau, Maurer, & Melnik, 2003), (Desai & Shah, 2011), (Cataldo & Ehrlich, 2012)). The case for additional research in the area software development has been called for by Herbsleb (2007), with further research relating to tacit knowledge, and the role which it plays in software development processes, called for by Ryan and O'Connor (2009), and Dingsøyr and Šmite (2014). As part of chapter four, a research model and methodology are outlined, including hypotheses development, based on discussions which have taken place in chapters two and three.

4 Research Model and Methodology

This chapter outlines a research model based on literature covered in chapter two and three. The initial sections of this chapter identify hypotheses, based on the aforementioned literature, with subsequent sections proposing a method of field research, to be carried out in a bid to ascertain the validity of the identified hypotheses. Eisenhardt (1989) has offered some advice regarding theory building from field research, and in doing so highlights the importance of some primary steps which should be taken into consideration, prior to entering field research:

1. *Definition of a research focus and identification of a priori knowledge.*
2. *The development of hypotheses and constructs.*
3. *Case study identification and selection of research instruments and protocols.*

In keeping with the definition of a research focus, the purpose of any case study is to address the primary research question. In this particular case, this relates to an investigation into *the relationship between system test complexity and tacit knowledge*. The use of a priori in the identification of constructs has been applied to good effect by Pee et al. (2010), prior to field research relating to knowledge sharing in information systems. This has been described as often important by Eisenhardt (1989), with claims that it helps shape the design of initial theory formation, often allowing researchers to measure constructs more accurately, and providing a firm empirical grounding if such constructs prove important as the research progresses. Initial a priori constructs provide the basis for hypotheses development, through a review of literature which has been covered in relation to the following topics:

- *The software development process and the role of software testing.*
- *Types of complexity which can potentially have an impact on the task of software testing.*
- *The relationship between tacit knowledge to the development process and in particular the relationship of tacit knowledge to software testing.*

Eisenhardt (1989) has stated that ideally, theory-building research is begun as close as possible to the ideal of no prior theory under consideration, and no initial hypotheses to test, a view endorsed by Urquhart (2010). According to Eisenhardt, that while this is impossible to achieve in practice, that researchers should strive to formulate research problems with important variables detailed, backed up by literature, but one should avoid thinking about specific relationships and theories. In keeping with the views of Eisenhardt (1989), and supported by an approach applied by Cataldo and Ehrlich (2011), the next step is the identification of relevant hypotheses, along with constructs which are identifiable through discussions, conducted as part of chapter two and three. In addition to detailing hypotheses, the subsequent sections deal with the other primary considerations of preparing for field research which have been previously been identified, namely *case study identification* and the *selection and the creation of instruments and protocols* (dealt with as part of a proposed approach to data collection). The focus and objectives of field research are also outlined, with approaches to each research stage discussed in detail, and ideal participants identified.

4.1 Research Objective

Research hypotheses which are developed as part of this chapter have been based on previously discussed literature. The objective of these hypotheses is to provide empirical evidence regarding the relationship between system test complexity and tacit knowledge. The important role which hypothesis development can play, when carried out prior to research, has been highlighted by Pee et al. (2010), Brown et al. (2011) and Cataldo and Ehrlich (2011). The role of software testing, discussed as part of chapter two, along with system test complexity and the concept of tacit knowledge, discussed as part of chapter three, are key to the development of the hypotheses. A detailed discussion on the hypotheses takes place in the following section.

4.1.1 Research Hypotheses

As part of a discussion of literature associated with the verification and validation of software, the importance of software test was emphasised. Views expressed were in keeping with the views of a number of authors ((Horgan & Mathur, 1996), (Eickelmann & Richardson, 1996), (Walter & Grabowski, 1999), (En-Nouaary, 1998), (Desai & Shah, 2011), (Holzworth, Huth, & deVoil, 2011). The strong relationship between complexity associated with aspects of the software development process, and knowledge, has been highlighted by Espinosa et al. (2007), Staats et al. (2010), Lu et al. (2011), Wang et al. (2012). In the software development overview section (2.4), fundamental aspects of development processes were outlined which are common to software development, Huo et al. (2004). These were highlighted as:

1. *Software specification and design:* The functionality and constraints associated with the software must be defined. This may take the form of requirements definition and software and system designs, or alternative approaches such as user stories, system metaphors, architectural spikes, and release planning.
2. *Software implementation:* In line with the requirements, goals and designs, the software must be produced. This can be a planned iterative development process, or a planned, sequential, development process.
3. *Software verification and validation:* The software must be validated to ensure it acts in accordance with customer requirements or standards. Code verification can take the form of static checks such as code reviews, inspections, and peer programming, or dynamic approaches such as software testing in the form of unit and system testing. Validation can take the form of customer feedback and acceptance testing.

Chapter two has outlined the main activities associated with software testing, a software verification technique, including the views expressed by Eickelmann & Richardson (1996), and Tsui and Iriele (2011) i.e. covering *test planning*, *test development*, *test execution*, *test failure analysis*, *test measurement*, and *test management*. Covered as part of chapter three, is the concept of tacit knowledge. The significance of tacit knowledge in software development environments has been emphasised by Ryan and O'Connor (2009), and Desai and Shah (2011). Central to the

first and second hypothesis, are the views of Nonaka and Von Krogh (2009), and Hedesstrom (2000).

Considering discussions which have taken place covering the work of McKeen et al. (1994) and Huo et al. (2004), Debbarma, et al. (2011) and Li, et al. (2011), the relationship between system test complexities associated with the system under test becomes important. Chapter three introduced the concept of inherent complexity associated with software systems ((Mumford, 1983), (Brooks F., 1995), (Lehman, 1996), (Lyytinen, Mathiassen, & Ropponen, 1998), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007), (de Silva & Balasubramaniam, 2012)). The first Hypothesis puts forward the premise that system testing is affected by complexity which is related to the system under test, and that most of the related knowledge does not lend itself to being made explicit.

Hypothesis 1 (H1):

The process of system testing (comprising of test case planning, test case development, *test case execution*, *test case fault analysis*, *test case measurement*, and *test case management*), is directly affected by complexity associated with the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit knowledge. It is also proposed that most of this tacit knowledge does not lend itself to being made explicit.

Andrade et al. (2013) have referred to the increasing complexity associated with software testing tasks. Brooks (1986) has made a distinction between essential complexity and accidental complexity associated with software engineering, with difficulties associated with the nature of software, being described as essentially complex, and difficulties associated with the production of software, being described as accidentally complex. The second hypothesis is concerned with the production of software, from the perspective of system testing. Debbarma, et al. (2011) have argued that there has been increasing complexity, along with the increasing size and performance demands of software systems, all of which demands more effective software testing. Hypothesis two proposes that such a relationship exists between complexity associated with system test testing, and the system under test. In contrast to

the knowledge associated with the system under test, it is proposed that a certain amount of knowledge relating to the process of system testing does actually lend itself to being made explicit.

Hypothesis 2 (H2):

That the process of system testing (comprising of *test case planning*, *test case development*, *test case execution*, *test case fault analysis*, *test case measurement*, and *test case management*), is affected by other sources of complexity, independent of the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit knowledge. It is proposed that much of this tacit knowledge does indeed lend itself to being made explicit.

The following section outlines a research strategy, which essentially details how one might validate the aforementioned hypotheses.

4.2 Research Strategy

This section attempts to align the research strategy with the research objectives. The previous section detailed hypotheses which are the basis for further investigation of the relationship between complexity and tacit knowledge associated with system testing. Also highlighted, and detailed in figure 4.1, there are two primary areas of focus regarding complexity and tacit knowledge, complexity and tacit knowledge relating to *system under test* and complexity and tacit knowledge relating to the actual *system testing.*

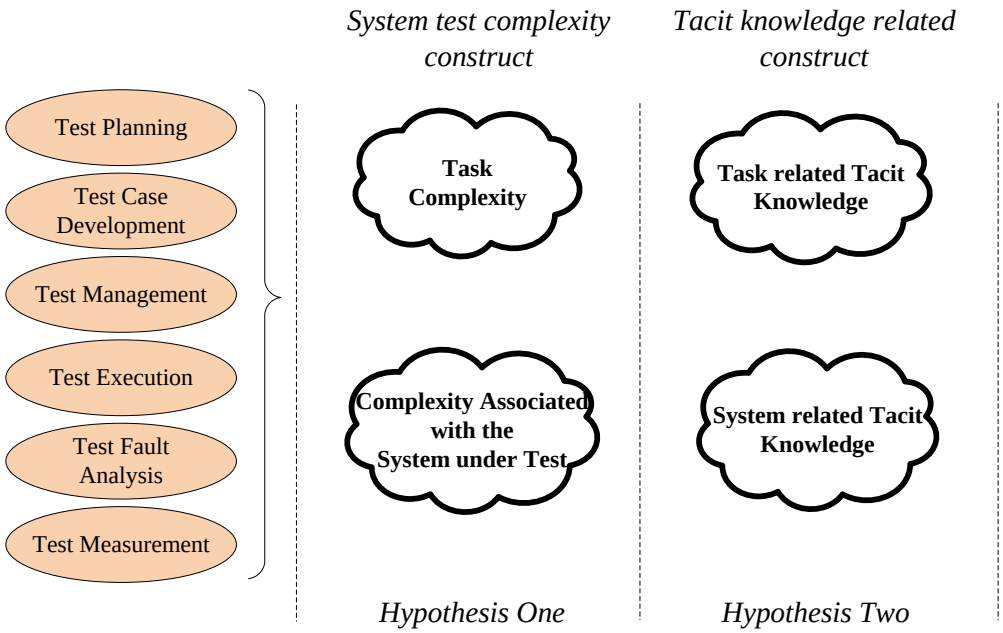


Figure 4.1: Research Model Constructs.

Detailed in figure 4.1 are:

- The functions or stages of system testing i.e. *system test planning*, *system test development*, *system test execution*, *test fault analysis*, *test measurement* and *test management*, as defined by Desai and Shah (2011).
- Complexity and tacit knowledge relating to both the system under test and the wider process of system testing. These two primary focus areas relate to the first and second hypothesis respectively.

The relationship between aspects of the model, detailed in figure 4.1, is proposed to be tested through field research. The following section offers potential research approaches, including the four assessment models as discussed by Wagner and Sternberg (1985) i.e. *the motivational*, *the critical incident*, *the simulation* and *the assessment center approaches.* The proposed approach to data collection (*creation of instruments and protocols)* is also discussed in the following section. The final section of the research strategy highlights the proposed interview questions.

4.2.1 Research Approach

This section discusses potential research approaches, with a case being made for what is perceived to be the most suitable approach. Authors such as Wagner and Sternberg (1985), Ryan and O'Connor (2009), Connelly et al. (2012) and Ahmad et al. (2012) have adopted more general interview approaches to field research. Four assessment models that may be applied through an interview approach to research have been proposed by Wagner and Sternberg (1985), *the motivational*, *the critical incident*, *the simulation* and *the assessment center*. These are discussed in more detail along with the *repertory grid technique*, developed by Ford et al. (1991), (used in the identification of tacit knowledge by Ryan and O'Connor (2009)), and the *grounded theory* method, developed by Glaser and Strauss (1967), (the application of which is referred to by Charmaz (1995), Martin et al. (2009), and Urquhart et al. (2010)).

1. The first approach, the *motivational approach*, attempts to increase understanding and predictability of real-world intellectual competence by considering the role of motives that drive and are satisfied by intellectual behaviour (such motives are referred to as n-Arch). Schüler et al. (2010) have described this technique as being based on the assumption that motives differ in strength and that these differences can explain behavioural differences.

2. The *simulation approach* attempts to highlight job competencies through work observation. The in-basket technique, developed by Frederiksen et al. (1957) is an example of this approach. This technique was defined to help measure and understand the skills associated with complex tasks, whilst also highlighting the problems and events associated with such tasks. This techniques also aids understanding of the decision making process which is central to task accomplishment. Wagner and Sternberg (1985) described this technique as consisting of providing the subject with a set of tasks, with performance being evaluated on accomplishment. It has been applied effectively by Sternberg et al. (1999) who employed this technique as a means of capturing and understanding the tacit knowledge associated with U.S. army commanders. The goal in that particular case was to make explicit tacit knowledge which could be then used

to train less experienced team members and also to formalise a method for tacit knowledge assessment.

3. Another framework suggested for tacit knowledge measurement is the *assessment center approach*. It is considered as another simulation approach by Sternberg (1999). Credited to Thornton and Byham (1982), it is a collection of various aspects of other approaches involving in-basket tests, interviews, and group discussions. In their execution of traditional aptitude and personality tests, Wagner and Sternberg (1985) have applied an assessment center approach, with performance appraisal consisting of summary judgements and ratings by groups of assessors.

The motivational approach is not as desirable in the context of this particular research, considering the objectives at hand. This research is primarily concerned with the relationship between tacit knowledge and system test complexity, as opposed to the motivational factors affecting system test complexity and tacit knowledge. Notwithstanding the benefits of the other approaches detailed i.e. the simulation approach and assessment centre approach, significant access to participants is required, either for observational reasons, in the case of the simulation approach, or in order to perform a collection of different approaches, required as part of the assessment centre approach. After taking into account the human resource cost of prolonged participant involvement, neither of these methods was deemed feasible for this research. Another downside to the simulation approach is highlighted by Sternberg (1999), who states that while this particular approach has the advantage of closely representing actual job performance, it is somewhat subjective as to what aspects of the job should be chosen to simulate, or how performance should be evaluated. Other more suitable methods discussed, which lend themselves to a more general interview approach, are *the critical incident technique*, *the repertory grid technique* and *the grounded theory approach*:

4. The *critical incident technique* is described as being based on research conducted on the Air Force during World War II by Flanagan (1954). Wagner and Sternberg have used the example of this technique which was later applied by McClelland (1976) to assess managerial competence. This method consisted of asking team members to detail several incidents which they handled

particularly well and several incidents which they handled particularly poorly. These detailed incidents are then analysed on a qualitative basis. This method is seen as a viable alternative to work observation but the validity of this approach is based on team members' willingness and ability to respond and the subsequent the qualitative analysis is sufficiently reliable. A similar approach has been used to good effect by Connelly et al. (2012), in a research effort to identify hidden knowledge.

5. The *repertory grid technique* of knowledge assessment is provided by Ford el al. (1991), Ryan and O'Connor (2009), and Cho and Wright (2010). This technique is founded on Kelly's (1955) theory of human understanding called the *Personal Construct Theory*. It has been proposed as a method for the identification and clarification of tacit knowledge by Jankowitz (2004) and Ryan and O'Connor (2009). According to Jankowitz there are four key elements:
    1. *The topic*
    2. *Constructs*
    3. *Elements*
    4. *Links*

    The repertory grid provides a two-way classification of information in which relationships are uncovered between a person's observations of the world, *elements*, and how they classify or make sense of those observations (via *constructs*). The central theme of the personal construct theory is that people are made up of contrasts rather than absolutes and a central premise of the theory is that every person's construct system is composed of a finite number of dichotomous or directly opposing constructs. The identification of constructs of a given topic is described as a very straight forward task, requiring the interviewee to be given plenty of examples of that topic, and analysing the results after they put those examples together. Repertory grids are described as an excellent method for structured interviewing, allowing the interviewee's viewpoint to be expressed with minimal contamination. It is also described as a method by which a stronger link can be made between qualitative data resulting from the repertory grid technique and quantitative research data.

6. *Grounded theory method* is defined by Glaser and Strauss (1967). This method is described as the discovery of theory from data which is systematically obtained through social research. Charmaz (1995) has provided an overview of grounded theory. Starting with individual cases, incidents or experiences, one develops more abstract conceptual categories, to synthesize, to explain and to understand data and ultimately to identify patterned relationships within accumulated data. Fundamentally, it is stated that grounded theories unite the research process with theory development. Urquhart et al. (2010) have claimed that this method offers well signposted procedures for data analysis, and potentially allows for the emergence of original and rich findings that are closely tied to data. This potential relationship between findings and the accumulated data can provide researchers with great confidence. Five main characteristics of the grounded theory method are outlined by Urquhart et al.:

    1. The main purpose of the theory is *theory building.*
    2. As a general rule, the researcher should make sure that their prior expertise does not lead them to pre-formulated hypotheses that their research then seeks to verify. Such preconceived ideas could hinder the emergence of ideas which should be firmly rooted in the data.
    3. Analysis and concept development are enabled through data collection and comparison, where data is compared against all existing concepts and constructs to see if it adds or enhances the knowledge regarding existing categories.
    4. The data collected is acquired by a method of theoretical sampling, where the researcher decides where to sample from next, based on analytical grounds.

    Urquhart et al (2010) have stated that studies in information systems have been criticised for having a relatively low level of theory development. Applications of the theory in the area of information systems (and other areas) have used grounded theory as method of coding data, instead of a method of generating theory. The authors felt that such an application of grounded theory limits the potential of the theory and the ultimate goal of the theory application should be as an enabler in the development of new theories. This view is backed up by

> Charmaz (1995) who has stated that the simultaneous activities of data-gathering and analysis, as part of grounded theory are explicitly aimed towards theory development.

The grounded theory has not been applied in this instance, even though successful application of this approach has been carried out by Martin et al. (2009) and Urquhart et al. (2010). The approach to this particular research would appear to be at odds with the main principles of grounded theory, as described by Urquhart et al. (2010), who states that as a general rule, the researcher should make sure that prior knowledge does not lead them to pre-formulated hypotheses. Section 4.2 outlines the pre-formulated hypotheses relating to this particular work and while, according to Urquhart (2010), aspects of grounded theory has been used successfully applied without staying true to the original theory, there are perceived to be other more suitable techniques available such as the critical incident technique, as devised by Flanagan (1954). The *critical incident technique* is an appropriate technique for use in this particular research because, as referred to by Butterfield et al. (2005) and Fitzgerald et al. (2008), this method has demonstrated its merit in the following aspects of research:

- *The identification of effective and ineffective ways of doing something, and also the identification of factors which either help or hinder.*
- *The collection of functional or behavioural descriptions of events or problems.*
- *The examination of success and failure.*
- *The determination of characteristics which are critical to important aspects of an activity or event.*

Give the benefit of the critical incident technique in the identification, collection and examination and determination of behaviours, events and activities, it would appear to be a suitable approach to identifying tacit knowledge, given that Ahmad et al. (2012) has referred to the measurement of *learning*, *thinking*, and *decision making skills* as considerations in the measurement of tacit knowledge, one of the primary focus areas for this research. Fitzgerald et al. (2008) have highlighted other benefits of such an approach, stating that the flexibility associated with the critical incident approach to case study research is a major benefit, with the method being most suited to one-on-

one interviews. Fitzgerald et al. (2008) have outlined the following key steps to performing critical incident based research:

1. Identification of general aims: pertinent research questions are described as being important prior to undertaking any type of research.
2. Planning: issues relating to participant selection, researcher familiarity with the research context, the methods of data collection, and the method of data analysis, are all considered important aspects of planning which should be considered prior to field research.
3. Data Collection: a number of key points are raised in relation to data collection. These are dealt with in more detail in the following section.

The following section proposes a suitable data collection model based on the chosen research approach.

4.2.2 Proposed Data Collection Model

As referred to in the previous section, McGrath (1984) has stated that any research ideally should ideally consider goals relating to the generalization of evidence over the population of actors, the precision of measurement of the behaviours under study, and the realism of the situation or the context of the research setting. What has been advocated by McGrath (1984), and Woodside (2009) is a balanced approach to evidence gathering. A fixed-point, survey questionnaire type approach, has been adopted by Pee et al. (2010), Hsu et al. (2011) and Akman et al. (2011). However such an approach is referred to as lacking in realism of context, and is deemed to be low in precision of measurement (McGrath, 1984). Similar concerns have been raised by Woodside (2009) (see figure 4.3), who states that there are four principle arguments against a questionnaire type approach to research:

1. *The difficulty with the translation of implicit thought to explicit thought* and further difficulty associated with the rating of such thoughts, as often required by a fixed-point survey type approach.

2. *The lack of suitability of fixed point constructs* such as questionnaires to feelings such as trust, perceived quality, and satisfaction.
3. *There is an assumed symmetrical relationship between independent and dependent variables*. This is described as not being necessarily truthful because of the possibility of alternative routes to a given outcome, often outside of the bounds of the questionnaire and therefore resulting in not being detailed.
4. *The unsuitability to measuring alternative complex relationship between dependent and independent variables*. This highlights the limitation of such approaches in measuring the unique contribution of each independent variable, and the variation in dependent variables.

The shortcomings of individual approaches to data collection such as a fixed-point survey approach or approaches such as a case study research approach are detailed in figure 4.2. Woodside (2009) highlights the following concerns relating to a case study approach:

1. Difficulties with researchers carrying personal and cultural value configurations implicitly into the field research thereby affecting judgments and statements.
2. Difficulties associated with "thick descriptions" relating to process in specific context. Such descriptions make a case for generalization beyond the immediate case.
3. Variability which may exist in different interpretations of verbal data relating to "thick descriptions", which are provided by participants.
4. Questionable relevance of the case study to other contexts given the absence of deductive theory or due to a small number of contexts to which the case study may have been applied.

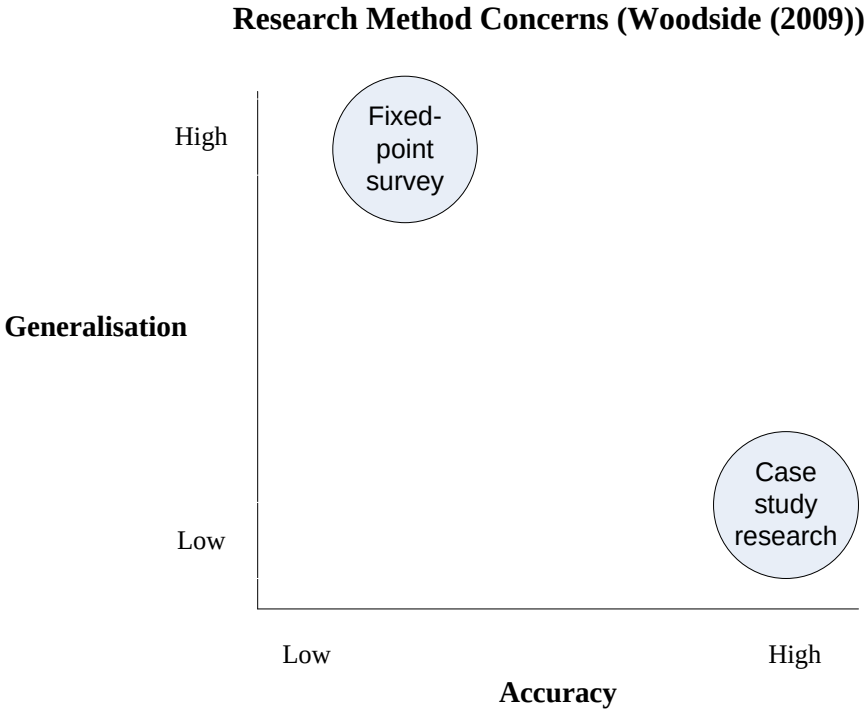


Figure 4.2: Research Method Concerns.

McGrath (1984) has held similar concerns, highlighting difficulties relating to precision of measurement when applying case study methods such as interviews. As an alternative to an independent questionnaire or case study approach, a more general interview approach, in with the previously discussed critical incident technique (used to good effect by Kaplan and Duchon (1988)), is proposed to be employed here. A general interview approach has been used to good effect by Ryan and O'Connor (2009), and helps address concerns addressed by Woodside (2009) and McGrath (1984), both of whom have questioned the ability of approaches such as fixed-point surveys, to measure alternative complex relationships, when used in isolation. The proposed approach also attempts to address the concerns expressed by Woodside and Baxter (2013), whereby a fixed point questionnaire approach is rejected, based on the inability of such an approach to provide the detail and accuracy necessary.

The following section provides the proposed interview questions.

4.2.3 Interview Questions

The importance of the identification of research aims via pertinent research questions is highlighted by Fitzgerald et al. (2008). In an effort to validate the research aims which have been outlined, and as part of the research strategy, interview questions have been identified which are outlined in this section. These have been categorised in table 4.1. The selected questions have been based on previous work carried out by Sternberg (2000), Chau et al. (2003), Espinosa et al. (2007), and Ryan and O'Connor (2009). Sternberg (2000) highlighted a number of sources of problem types relating to practical intelligence and tacit knowledge, but more relevant to this work, is the identification of critical areas in a development process which depend on knowledge transfer (Chau, Maurer, & Melnik, 2003). As well as the identification of system test complexity and the relationship to tacit knowledge, the influence of factors relating to *geographical distribution of teams, software development characteristics*, *the role and experience of the participant*, *knowledge management*, *test environment characteristics*, are also of interest. Also key is the identification of sources of complexity which may be due to common development practices in operation. Relevant development characteristics were described in section 2.1, as part of an overall discussion on development models.

The questions detailed in table 4.1 are designed to preserve the anonymity of the subject to encourage honest and open responses. A quantitative element to the questions has been included via the request for appropriate ratings. These ratings help ascertain the significance of a relationship, with a *likert scale* is being used (on a scale of 1-7, where one highlights a weak or non-existent relationship and 7 highlighting a very strong relationship). Also included in this table is detail relating to how the output of each question is expected to feed into further analysis. It's expected that questions 3, 4, and 5, will provide the basis for quantitative analysis.

Number	**Questions**	**Relationship to Analysis**
1.	Your team consists of co-located (locally based) team members? Yes/No	Qualitative
2.	How would you describe your current job? i.e. Manager, lead	Qualitative

	engineer, or engineer.	
3.	What level of experience do you have which is relevant to the current role (number of years)?	Quantitative / Qualitative
4.	Have you encountered complexity associated with the following system testing tasks, whereby there was insufficient or an absence of necessary documented knowledge to enable a satisfactory solution? Please elaborate with reference to the following stages of system testing. Please relate your experiences to a previous project, which might be typical of your experience: 1. *Test Case Planning* (what needs to be tested and how it should be tested, given available resources) 2. *System test Development* (development of a test environment and test suites) 3. *Test Suite Execution* (execution of test cases, from a manual or automated test perspective 4. *Test Case Fault Analysis* (debug and root cause analysis of issues which arise after test execution) 5. *Test Case Measurement* (assessment of overall system quality) 6. *Test Case Management* (management of the test environment, resources, etc.)	Quantitative / Qualitative
5.	Have you encountered complexity in execution of your job due to insufficient knowledge relating to the actual system under test? (Such complexity may relate to system functionality or system deployment.) How would you rate the relationship (if any), of the system under test, to complexity and tacit knowledge? Relationship to complexity rating: Relationship to tacit knowledge rating:	Quantitative / Qualitative
6.	Could you please provide examples of other sources of complexity associated with your job? If yes, please provide an example: How would you rate the relationship (if any), of such sources of complexity to complexity and tacit knowledge?	Qualitative

	Relationship to complexity rating: Relationship to tacit knowledge rating:	
7.	How would you describe the communication of development specifications i.e. are they mainly communicated formally e.g. in the form of user stories or functional specifications, or informally via verbal communication? Are they communication on an incremental basis? How would you rate the relationship (if any), of such specifications to complexity and tacit knowledge? Relationship to complexity rating: Relationship to tacit knowledge rating:	Qualitative
8.	How would you describe communication with the development team? Is it on a regular basis, starting from the system test planning/user story development phase? How would you rate the relationship (if any), of such development communication to reducing complexity and how would you rate the tacit knowledge associated with such knowledge? Relationship to complexity rating: Relationship to tacit knowledge rating:	Qualitative
9.	Would you agree that there is strong dependency within your team, on the knowledge of other team members (please ignore if irrelevant)? How important is the availability of such knowledge? How would you rate the relationship (if any), of such knowledge in reducing complexity associated with your job. How also would you rate the relationship to tacit knowledge necessary for your job? Relationship to complexity rating: Relationship to tacit knowledge rating:	Qualitative

10.	How familiar are you of the work of any other team members who work independently from you? How would you rate the relationship (if any), of such work to complexity and tacit knowledge? Relationship to complexity rating: Relationship to tacit knowledge rating:	Qualitative
11.	Have you encountered specific gaps in available knowledge which has affected your ability to excel in your job? How would you rate the relationship (if any), of such gaps in available knowledge to complexity and tacit knowledge? Relationship to complexity rating: Relationship to tacit knowledge rating:	Qualitative

Table 4.1: Research Questionnaire.

As part of the interview stage, additional questions have been identified in table 4.1, relating to previous experiences of system text complexity and tacit knowledge, and an additional question relating to team composition. The importance of a balanced approach to evidence gathering has been referenced by McGrath (1984), and Woodside (2009). The next section deals with putting into practice the outlined research approach. The criteria for suitable research candidates are identified, with a sampling strategy outlined.

4.3 Research Design

The previous sections dealt with the research objectives and the proposed research strategy. This section deals with research design and preparatory stages to be considered prior to field research i.e. *case study identification and selection.* Eisenhardt (1989) and Fitzgerald et al. (2008) have described population selection as

an important consideration for case selection, enabling the definition of a set of entities from which research samples can be drawn. The following section discusses such considerations, following on with a subsequent discussion on appropriate data collection approaches.

4.3.1 Case Study Selection

Consideration of population selection can provide control over environmental variation as well as enabling the definition of limits for the analysis of findings (Eisenhardt, 1989). As suggested by Pettigrew (1988), and observed by Eisenhardt, it makes sense to select cases such as extreme situations and polar types in which the process of interest is "transparently observable". Therefore, in line with this view, cases are chosen on the likelihood that they have the potential to replicate or extend emergent theory. This would also be in keeping with goals, highlighted by McGrath (1984), who states that as part of the data collection process, that the following goals should be considered:

1. The *generalization of the evidence* over the population of actors.
2. The *precision of measurement* of the behaviours under study. Also mentioned is the precision of control over extraneous facets or variables that are not being studied.
3. The *realism of the situation* or the context of the research setting. This is referred to as relating to the context to which you want your evidence to refer.

In addition to the above guidance, fig 4.3 highlights additional, desirable characteristics, which are being sought regarding potential research organisations.

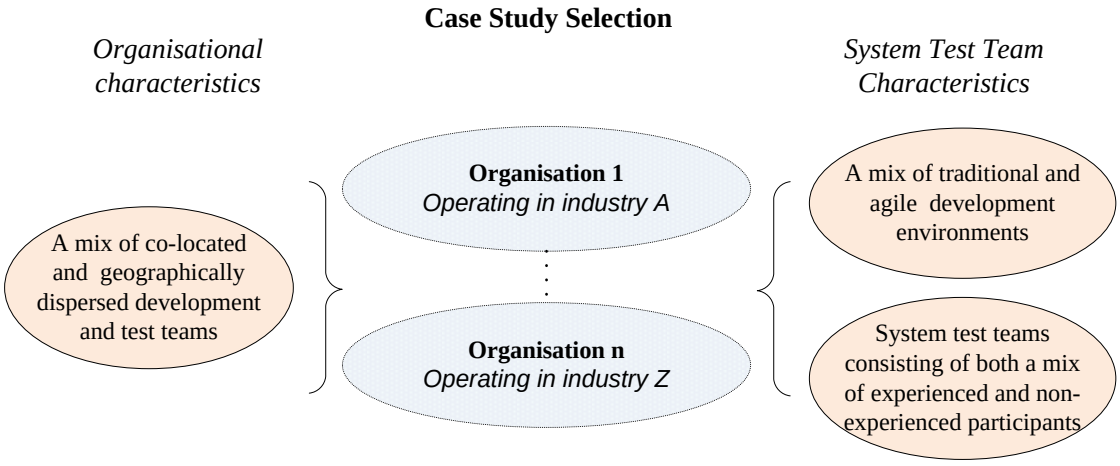


Figure 4.3: Case Study Selection Criteria.

As detailed in figure 4.3, the selected organisations selected organisations should have the following desirable attributes:

- *Independent test teams, consisting of a mix of co-located and geographically dispersed development and test employees.*
- *A mix of experienced and inexperienced system testers.*
- *There was a mix of traditional and agile software development environments being employed across the chosen organisations.*

The following four organisations were selected because they displayed the attributes and circumstances that made them suitable case studies for this research project:
EMC Corporation (EMC²), SQS Software Quality Systems AG, Delaware Life, and CoreHR:

1. *EMC²* is an American multinational corporation, headquartered in Hopkinton, Massachusetts, United States. They offer data storage, information security, virtualization, analytics, cloud computing and other products and services that enable businesses to store, manage, protect, and analyse data. EMC was founded in 1979, has grown to over 60,000 employees, and is currently considered one of the world's largest providers of data storage systems.

The chosen system test teams were interviewed at the EMC office at Ovens, Cork, Ireland. They displayed the following characteristics:

- There were 37 participants interviewed in total. This involved four different projects and eleven different system test teams. 1 interview was discounted due to the participant's focus on automation rather than on the various stages of the system test process.
- 27 system testers were operating in a traditional software development environment, with 10 system testers operating in an agile software development environment.
- The system test experience ranged from just over 1 years' experience, to 23 years' experience.
- The teams work on a daily basis with system test colleagues and developers based in Boston, MA, US, and in Bangalore, India.

2. *SQS Software Quality Systems AG* is a consultancy company based in Cologne, Germany. The company describes itself as the largest independent provider of software testing and quality management services. The SQS Group was founded in Cologne in 1982 and has around 3,800 employees. SQS has offices in 13 countries covering Europe, Africa, Asia and North America.

   The SQS participants all work in the SQS Dublin office. This set of participants were characterised by the following characteristics:

   - There were 5 participants interviewed in total. These participants worked on five different projects, involving different system test teams.
   - 4 system testers were operating in a traditional software development environment, with 1 system tester operating in an agile software development environment.
   - The system test experience ranged from just over 2 years' experience, to 15 years' experience.
   - The participants all had regular experience of working with remote development teams.

3. *Delaware Life*, a leading provider of annuity and life insurance products, is based in Boston, in the United States. Delaware Life was established in August 2013 in connection with the purchase by Delaware Life Holdings, LLC, of the domestic U.S. annuity business and certain individual life and corporate markets insurance businesses, from Sun Life Financial Inc. After the acquisition, a section of Sun Life Financial employees were transitioned over to Delaware Life, including a subset of the Sun Life employees in Waterford. Sun Life has had a strong presence in Waterford for over 15 years, part of a wider global workforce of more than 15,000 employees and 12,000 advisors. More than 500 employees transitioned in total from Sun Life to Delaware Life.

   The Delaware Life participants all work in the Waterford office. They displayed the following characteristics:

   - There were 6 participants interview in total, operating on different aspects of a migration project. The migration project consisted of moving data and applications from the original company, Sun Life, to the new Delaware life organisation.
   - All system testers were operating in an agile software development environment.
   - The system test experience ranged from just over 6 years' experience, to 18 years' experience.
   - The participants all had regular experience of working with remote teams, located in the head office in Boston, MA, US.

4. *CoreHR* has been providing HR and Payroll software solutions to organisations in the UK, Ireland, and Europe for over 30 years. The organisation's headquarters are in Cork, Ireland, with offices also located in London, Dublin and Kilkenny. CoreHR has more than 200 employees at present.

   The CoreHR participants work in the Ballincollig office in Cork, and the Kilkenny office. They displayed the following characteristics:

   - There were 4 participants interview in total, working on four different projects with CoreHR.

- 3 system testers were operating in a traditional software development environment, with 1 system tester operating in an agile software development environment.
- The system test experience ranged from just over 2 years' experience, to 12 years' experience.
- The participants all had experience of working with remote colleagues.

The test teams from the four participating organisations were in total responsible for testing ten different systems. The test teams varied in team sizes from four testers to ten testers, with all teams operating some level of geographically dispersion between team members. Sixty two interviews were conducted in total across the four participating organisations. A preference was expressed for face to face interviews to be facilitated, where possible. There was a split in the employed development methodology, across the different development environments involved. Those that were applying a traditional approach to software development were carrying out the process of specification, design and implementation, prior to any significant system testing taking place, whereas those teams which were adopting an agile approach to software development were in line with some, if not all, of the following common characteristics of an agile software development approach, detailed in chapter 2:

1. *The processes of specification, design and implementation ran concurrently*. Detailed system specification, and design documentation are minimised or generated automatically by the programming environment used to implement the system. Usually only the most important characterics of the system are defined as part of the user requirements document.
2. *Systems are developed in a series of increments*. End-users and other system stakeholders are involved in specifying and evaluating each increment after which changes and new changes are proposed to be catered for in subsequent increments.
3. *System user interfaces are often developed using an interactive development*. This approach enables the quick creation of interface designs.

As a whole, the selected test teams displayed the following primary characteristics:

1. The test teams from the four participating organisations were in total responsible for testing 10 different systems.
2. The test teams varied in team sizes, from four testers to ten testers, with all teams operating with some level of geographically dispersion between team members.
3. 62 interviews were conducted in total across the four participating organisations (one was discounted due to a lack of participant exposure to all stages of system testing).
4. Experience of the participants varies from 1 years' experience to some participants with greater than 20 years' experience.
5. There was a variation in the employed development methodology, across the different development environments involved, with some teams operating in a traditional development environment, and some operating in an agile development environment.

The following section discusses the participating teams in more detail, highlighting also the proposed approach to data collection.

4.3.2 Sampling Strategy

Fitzgerald et al. (2008) have highlighted the following key points relating to data collection:

1. Observations should be reported and recorded for future analysis.
2. Such observations should be recorded as close to the time they occurred as possible, thus encouraging the accuracy of findings.
3. Central to the concept of the critical incident approach is the concept of trust, both in the accuracy of the reporting of the observer and between the observer and the participants. Trust between the observer and the participant usually requires a guarantee of anonymity for the participant.
4. Reports can be made as part of individual or group interviews, through questionnaires or through record forms. The collection method is stated as depending on choice such as participant availability, and the research subject

etc. The best option is described as being the individual interview approach, allowing for the best explanation of the aims of the study and clarification of ambiguities in the reports.

5. The number of required reports is described as something which is difficult to determine in advance, often demanding that sampling continue until a saturation point is reached whereby no further samples contribute any model influencing information for analysis.

Figure 4.4 outlines the data collection and analysis stage of the research. The potential repetitive nature of the data collection process is highlighted.

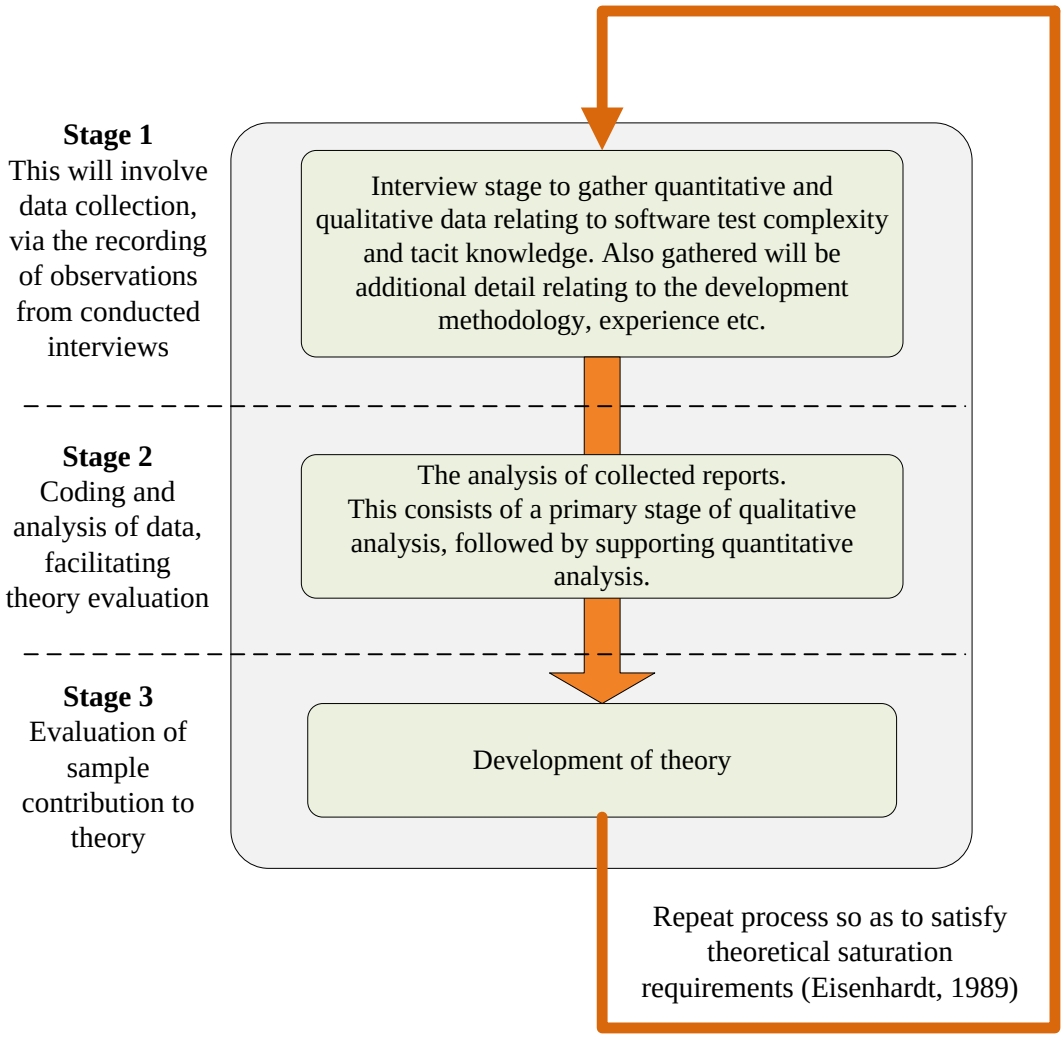


Figure 4.4: Stages of Data Collection.

As previously outlined, the primary goal of *stage one* of data collection, the interview stage, is the collection of data relating to system test complexity, and the relationship between system test complexity and tacit knowledge. Data collected relating to personal, organisational or environmental factors also form part of this stage. *Stage two* relates to the coding and analysis of data collected from stage one. *Stage three* evaluates the information collected, and thus facilitates a decision relating to the contribution of the samples which have been collected to date, with the eventual goal of field research ultimately being theory development. The next section provides an overview of the quantitative analysis, which was also conducted.

4.3.3 Data Analysis

The qualitative data relates to the output of the series of interviews which were conducted as part of this research. The method of data collection for this research is a collection of interviews, a similar technique to that conducted by Ryan and O'Connor (2009). Flanagan's critical incident technique has been employed, a technique which has been used by Kaplan and Duchon (1988), delivered via a series of open questions. The field research activities can be classified as having one of three objectives:

1. *Data collection*. This was achieved by carrying out a number of selected interviews. Data collected relating to personal, organisational or environmental factors also form part of this stage.
2. *The coding and analysis of collected data*.
3. *The evaluation of analysed data*, facilitating a decision to be made relating to the contribution of the samples which have been collected to date.

In addition to the qualitative data which was retrieved as a result of the 62 interviews conducted, there was also a quantitative aspect to these interviews. This is much in keeping with the views of Casti & Karlqvist (1986), who investigated the characteristics, influences and effects of complexity, in an ultimate attempt to reduce its effects. The quantitative analysis was carried out on way of variance based, partial least squares approach to structural equation modelling (PLS-SEM), using the

SmartPLS application, Ringle et al. (2005). This was used primarily because of its suitability for theory development, Hair et al. (2011). Further, confirmatory analysis of data is also proposed to be carried out by way of a covariance approach, using the Lisrel application, with analysis relating to indicator correlation proposed to be carried out using the IBM SPSS application. The SmartPLS has been shown some support, Lowry & Gaskin (2014). There were two primary features associated with SmartPLS algorithm, which are used for analysis:

1. The bootstrapping procedure involves taking a large number of subsamples (i.e., bootstrap samples) being drawn from the original sample with replacement (each time an observation is drawn at random from the sampling population, it is returned to the sampling population before the next observation is drawn). This confidence interval is derived from the t-statistic values, available as an output from the bootstrap procedure.
2. The PLS algorithm was used to calculate standardised regression coefficients between variables, providing an indication of the positive or negative relationship which may exist between the variables. Such relationships are referred to in the next section.

The primary constructs of *system test related complexity* and *system test related tacit knowledge* are reflected by six indicators. These six indicators reflect complexity and tacit knowledge, as they relate to the six different functions (stages) associated with system testing:

1. *System test planning*
2. *System test development*
3. *System test execution*
4. *System test fault analysis*
5. *System test measurement*
6. *System test management*

These indicators are correlated, thus making the variables reflective as opposed to formative. To measure the system test complexity and system test related tacit knowledge, participants were asked to rate the level of system test complexity, and

system test tacit knowledge, associated with each of the aforementioned indicators. The ratings were based on a seven point likert scale. Cenfetelli and Bassellier (2009) have provided some guidance for interpreting results associated with formative constructs. The following issues and guidance are provided:

1. *Muiticollinearity*. When excessive collinearity exists between indicators (multicollinearity), this introduces the potential for unstable indicator weights. An investigation into bivariate correlation between indicators and constructs should be performed. It is advised that variance inflation factors (VIF) should be assessed to determine whether multicollinearity is an issue. Any excessive overlap between indicators may be rectified by the removal of the offending indicators but consideration should also be given as to the effect the removal would have on the overall the meaning of the construct.

There is recognition of the role which multicollinearity can play in destabilising a model, Diamantopolous et al. (2008), Marciniak et al. (2014). O'Brien (2007) has described VIF (and tolerance), as being based on the proportion of variance which any one particular indicator, associated with a construct, shares with other independent indicators, associated with the same construct. As part of guidance to avoid multicollinearity, Kim et al. (2010) have recommended that formative indicators should cover the entire domain space of a construct, should be designed to avoid sharing a common theme, and therefore should not be interchangeable. There is an alternative view, that multicollinearity must be acknowledged as an accepted consequence in certain circumstances, and that it may be a difficult task to separate influences of the indicators associated with a particular construct (O'Brien, 2007).

Other concerns and guidelines raised by Cenfetelli and Bassellier (2009) involving indicator assessment are:

1. *The number of indicators*. With an increase in the number of indicators which determine a formative construct, there is an increased likelihood of some low or insignificant indicators. In the case of there being a large number of indicators, it is advised that steps such as the introduction of multiple indicators, the creation of second-order constructs can take place. In the

absence of the aforementioned steps being taken, there should be at least a discussion on the absolute contribution of the indicators.

2. *The co-occurrence of both negative and positive indicator values*. Negative values may be as a result of suppressor effects, whereby there is more variance between indicators than with the formatively measured construct. Thus investigation should be carried out as to the presence of suppressor effects. An investigation of bivariate correlation should also be carried out. One step which can be taken in the case of the presence of both positive and negative indicator weightings is the removal of indicators. This is providing the indicator is both acting as a suppressor and there is evidence of bivariate correlation also existing. An indicator with a significant negative weight, and with a positive bivariate correlation, should be interpreted as having a negative effect on other indicators.
3. *The absolute versus relative indicator contributions*. Indicators which have a relatively small contribution, in comparison to other indicators, may still have an important contribution, if that indicator is assessed independently from other indicators. Bivariate correlations should be determined to assess the contribution of independent variables. In such a case, where an indicator has a low, relative, contribution and a high bivariate correlation, the indicators importance should be recognised. If the indicator has both a low relative contribution and a low bivariate correlation, then the continued inclusion of such indicators becomes questionable.

Kim et al. (2010) have addressed some of the concerns highlighted in the previous points. In the case of formative constructs, the number of indicators can vary but the indicators should cover the entire domain of the construct and, and should avoid sharing a common theme which makes them interchangeable. Concern has been raised regarding activities relating to either the elimination of individual indicators relating to a construct, and the combination of indicators ((Kim, Shin, & Grover, 2010), (Diamantopoulos, Riefler, & Roth, 2008)). The aforementioned authors suggest caution against the removal of indicators, because it may result in an unexpected change in the overall meaning of the construct. Similarly, caution is advised regarding the impact of combining indicators, which may be carried out in an effort to increase indicator contributions.

Two additional concerns have been raised by Cenfetelli and Bassellier (2009), relating to the validity assessment of formative indices:

4. *Nomological network effects and construct portability*. Some degree of change in indicator weights should always be expected as the estimation of a formatively measured construct depends on other constructs in the model, but large changes are deemed to imply a lack of portability and thus threaten the generalizability of the interpretation of a given indicators contribution and so also the interpretation of the results of a model. An example is provided, that if a formative indicator weight which changes from being a large value in one nomological network, to a small value in another that would make the interpretation of its importance difficult to gauge. MIMIC/redundancy analysis is proposed as one method which can be used to assess the likelihood of interpretational confounding and an evaluation of the structural misspecification and the relevance of the choice of outcomes.
5. *The choice of technique*. If using a PLS technique, or if excluding construct error while using CB techniques, consideration must be given in interpreting results, to the potential inflation in weights.

In response to concerns relating to the nomological network effects and construct portability, serious consideration should be given to the assessment of indicator validity ((Edwards & Bagozzi, 2000), (O'Brien, 2007), and (Kim, Shin, & Grover, 2010)). Similar to the concerns to those raised relating to nomological network effects and construct portability, have been raised by Kim et al. (2010), who highlight *interpretational confounding* and the *external consistency of data* as being two aspects which should be examined in some detail. Interpretational confounding and external consistency are issues which may be faced as a result of incorrectly specified formative models. To deal with the effects of these issues, the importance of the pre-examination of data is emphasised as being particularly important in the case of formative indices, Kim et al. (2010). One approach which is recommended to identify the existence of interpretational confounding, is the comparison of both correctly, and deliberately incorrectly specified models. Issues associated with external consistency are recommended to be investigated by a review of the correlation between the formative indicators of a construct and the measures of a dependent construct. The

choice of technique and possible weight inflation should be also taken into consideration due to the application of a PLS techniques.

The following section provides an over of the research model which has been discussed in this chapter, including the research objective, research strategy and research design.

4.4 Summary of the Research Model

In line with the views of Eisenhardt (1989), the following points were taken into consideration, prior to entering field research:

1. *Definition of a research focus and identification of a priori knowledge.*
2. *The development of hypotheses and constructs.*
3. *Case study identification, and the selection and research of instruments and protocols.*

The use of a priori knowledge in the identification of constructs has been applied to good effect by Pee et al. (2010), prior to field research relating to knowledge sharing in information systems. A priori constructs have also been used to good effect in this research case. This provides the basis for hypotheses development, through a review of literature which has been covered as part of chapters two, and three, relating to the following topics:

- *The software development process and the role of software testing.*
- *Types of complexity which can potentially have an impact on the task of software testing.*
- *The relationship of tacit knowledge to the development process and in particular the relationship of tacit knowledge to software testing.*

In line with the second step, as outlined by Eisenhardt (1989), subsequent sections of this chapter deal with the development of hypotheses and constructs. The first hypothesis was developed taking into account of the views of authors such as McKeen et al. (1994), Huo et al. (2004), Debbarma, et al. (2011) and Li, et al. (2011), relating to task complexity. The views of others relating to the significance of inherent complexity, ((Mumford, 1983), (Brooks F., 1995), (Lehman, 1996), (Lyytinen, Mathiassen, & Ropponen, 1998), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007), (de Silva & Balasubramaniam, 2012)), were also acknowledged, as were the views of Ryan and O'Connor (2009), Desai and Shah (2011), Nonaka and Von Krogh (2009), and Hedesstrom (2000), regarding tacit knowledge. This hypothesis puts forward the premise that system testing is affected by complexity related to the system under test, and that most of such knowledge does not lend itself to being made explicit.

The second hypothesis is based on the work of authors such as Andrade et al. (2013), and Brooks (1986), with a distinction being made between essential complexity and accidental complexity associated with software engineering. Hypothesis two proposes that such a relationship exists between complexity associated with system test testing, and the system under test. In contrast to the knowledge associated with the system under test, it is proposed that a certain amount of knowledge relating to the process of system testing actually lends itself to being made explicit.

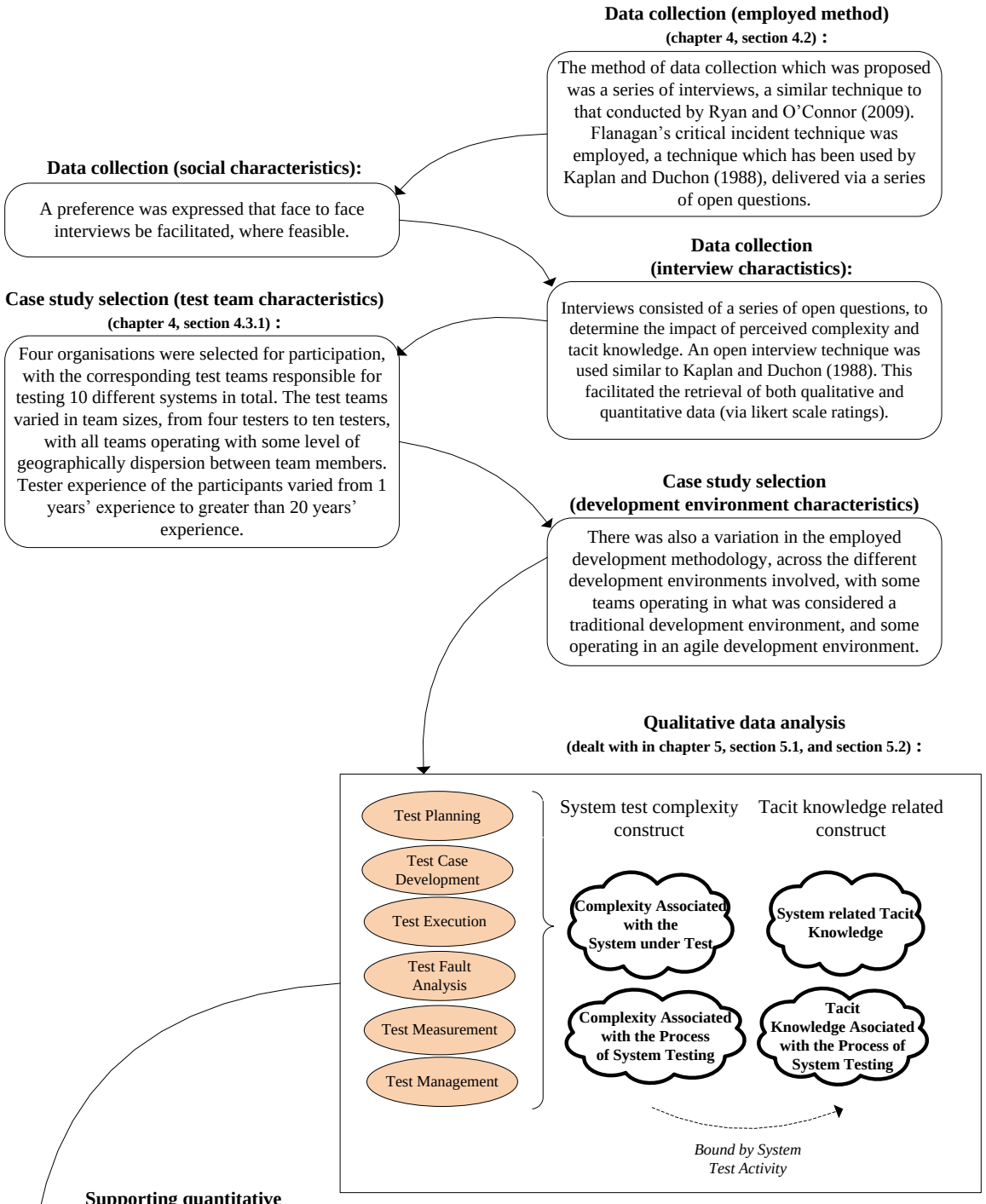
Rather than the identification of the difficulties and complexity which software testers face as a technological issue, some authors emphasise the importance of human factors, such as skill, experience, and management, in the achievement of software development goals ((Guinan et al., 1998), (Espinosa, 2007)), and their particular relevance in the achievement of software testing goals, (Martin, Rooksby, Rouncefield, & Sommerville, 2007). The link between tacit knowledge and experience has been made by both Polanyi (1966), and Nonako and Van Krogh (2009). The importance of experience has been emphasised by Crispin and Gregory (2009), and Desai and Shah (2011).

In summary, the two hypotheses which were proposed are:

1. The process of system testing (comprising of *test case planning*, *test case development*, *test case execution*, test case fault analysis, test case measurement, and *test case management*), is directly affected by complexity associated with the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit knowledge. It is also proposed that most of this tacit knowledge does not lend itself to being made explicit.

2. That the process of system testing (comprising of test case planning, *test case development*, *test case execution*, *test case fault analysis*, *test case measurement*, and *test case management*), is affected by other sources of complexity, independent of the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit knowledge. It is proposed that much of this tacit knowledge does indeed lend itself to being made explicit.

To investigate the identified hypotheses, the proposed method for data collection which was a series of interviews, a similar technique to that conducted by Ryan and O'Connor (2009). Flanagan's critical incident technique was employed, a technique which has been used by Kaplan and Duchon (1988), delivered via a series of open questions. The proposed data collection method, which consisted of a combination of a quantitative and qualitative approach, and a variety of open questions, relating to the qualitative aspect of the interview, would appear to be very much in keeping with the law of requisite variety (Ashby, 1956), whereby the fact that evidence is being sought from a variety of perspectives, relating to both tacit knowledge and complexity, demanded a certain variety in the research approach. Figure 4.5 provides an overview of the proposed research model and methodology in practice.



The quantitative data associated with the organisations in question
(C = complexity; T = tacit knowledge; 1.1 = test planning...1.6 test management; 2 = system under test)

Organisation	Industry	values	Cx1.1	TKx1.1	Cx1.2	TKx1.2	Cx1.3	TKx1.3	Cx1.4	TKx1.4	Cx1.5	TKx1.5	Cx1.6	TK1.6	Cx2	TKx2
A	Enterprise Storage	Average ratings	5.2	5.0	5.1	5.0	3.3	3.2	5.3	5.6	2.6	2.5	3.5	3.4	5.8	5.8
B	Test consultancy	Average ratings	6.0	6.0	5.3	3.8	4.3	4.0	5.0	5.3	5.0	4.8	2.3	2.5	6.0	6.5
C	Life Assurance	Average ratings	5.8	5.7	6.0	6.0	3.5	3.8	5.3	5.3	2.5	2.7	3.3	3.3	6.5	6.5
D	Payroll	Average ratings	5.5	5.5	4.3	4.5	4.0	4.3	4.5	4.5	2.5	2.5	2.0	2.0	6.0	5.5
		Standard deviation	0.36	0.40	0.72	0.94	0.43	0.45	0.38	0.48	1.23	1.09	0.76	0.67	0.30	0.50

Figure 4.5: Summary of Research Model and Methodology

Figure 4.5 makes reference to the interviews and interview questions, which are a key aspect of the data collection approach. The interview questions are detailed in section 4.2.3. The selected questions have been based on previous the work of numerous authors, detailed in chapters two, three, and four, some of which has been discussed in brief in the previous section. The following section provides an overview of conclusions which can be drawn from the analysis of the research data, including quantitative data which was also analysed.

Also referenced in figure 4.5, is a brief overview of the case study selection details, which had the following characteristics in detail:

- Four organisations were selected for participation, with the corresponding test teams responsible for testing ten different software systems in total.
- Sixty two participants were identified in total, involving test teams from the four selected organisations (one was discounted due to a lack of tester exposure to all stages of system testing). A preference was expressed that face to face interviews be facilitated.
- The test teams varied in team sizes, from four testers to ten testers, with all teams operating with some level of geographically dispersion between team members.
- Experience of the participants varied from one years' experience to greater than twenty years' experience.
- There was also a variation in the employed development methodology, across the different development environments involved, with some teams operating in what was considered a traditional development environment, and some operating in an agile development environment.

The bottom of figure 4.5 highlights the role of the quantitative data, in supporting the qualitative analysis. Presented are the average figures for the quantitative responses relating to questions 4 and 5 (as detailed in section 4.2.3). These questions specifically relate to complexity and tacit knowledge associated with the process of system testing (Cx1.x and Tx1.x), and complexity and tacit knowledge relating to the system under test (Cx2 and Tx2). These results are discussed in detail as part of chapters five and six.

5 Field Research

As proposed in chapter four, the method of data collection employed at the four organisations was a series of interviews, a similar technique to that conducted by Ryan and O'Connor (2009), and Connelly et al. (2012). Sixty one interviews were analysed in total across the four participating organisations, of which fifty three were conducted face to face, and eight were conducted remotely via teleconference. One additional interview was discounted because the participant was solely involved in development and maintenance of the test environment, and therefore had no exposure to test suite execution, test fault analysis, and test measurement aspects of system testing. The interviews were conducted during the time period between the 18th of October, 2013, and the 28th of March, 2014. There was a wide variation of participant experience across the organisations concerned, as detailed in table 5.1.

Participant Experience	Mean	Minimum	Maximum
Total employees (n=61)	8.16	1	23
< 10 years of experience (n=39)	4.79	1	9
>= 10 years of experience (n=22)	14.14	10	23

n = the number of samples;

Table 5.1: Breakdown by Participant Experience.

There was also a split in the employed development methodology, as detailed in table 5.2. There were 41 candidates who considered the applied development methodology as being traditional in nature, with 20 candidates considering the adopted approach as being an agile development approach.

Development Methodology	Number of Samples
Traditional Development Methodology	41
Agile Development Methodology	20

Table 5.2: Breakdown by Employed Development Methodology.

Those that were applying a traditional approach to software development were carrying out the process of specification, design, and implementation, prior to any significant system testing taking place, whereas those teams which were adopting an agile approach to software development were in line with some if not all of the common characteristics associated with an agile software development approach, namely concurrent specification, design and development stages, and the adoption of an incremental development approach.

This section provides an overview of the qualitative and quantitative analysis. This is conducted in sections 5.1, 5.2, and 5.3. As stated at the start of the analysis section, we are ultimately concerned with validation of the hypotheses as outlined in chapter 4. The two hypotheses which were outlined are:

1. The process of system testing (comprising of *test case planning*, *test case development*, *test case execution*, *test case fault analysis*, *test case measurement*, and *test case management*), is directly affected by complexity associated with the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit knowledge. It is also proposed that most of this tacit knowledge does not lend itself to being made explicit. The first hypothesis is primarily concerned with sections 5.1.1, and 5.1.2.

2. The process of system testing (comprising of test case planning, *test case development*, *test case execution*, *test case fault analysis*, *test case measurement*, and *test case management*), is affected by other sources of complexity, independent of the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit knowledge. It is proposed that much of this tacit knowledge does indeed lend itself to being made explicit. Relevant analysis associated with the second hypothesis, was covered as part of sections 5.1.3 and 5.1.4.

A number of relationships were evident from analysis carried out in previous sections. In line with the first and second hypotheses highlighted above, a distinction was been

made between complexity which is associated with the system under test, and complexity associated with the process of system testing (related to the process of system testing but excluding the system under test in practice). A similar distinction has been made between tacit knowledge associated with the system under test, and tacit knowledge associated with the process of system testing.

The following section provides analysis of data collected from field research interviews. Interviews were initially recorded and then transcribed from tape to facilitate detailed analysis of the various sentiments which were expressed by participants. Ultimately the analysis is aimed at evaluating the aforementioned hypotheses detailed.

5.1 Coding and Analysis of Data Relating to the First Hypothesis

The first hypothesis proposes that there is a positive relationship between *complexity associated with the system under test* and the relationship to *tacit knowledge.* This has been coded and categorised in sections 5.1.1, and 5.1.2.

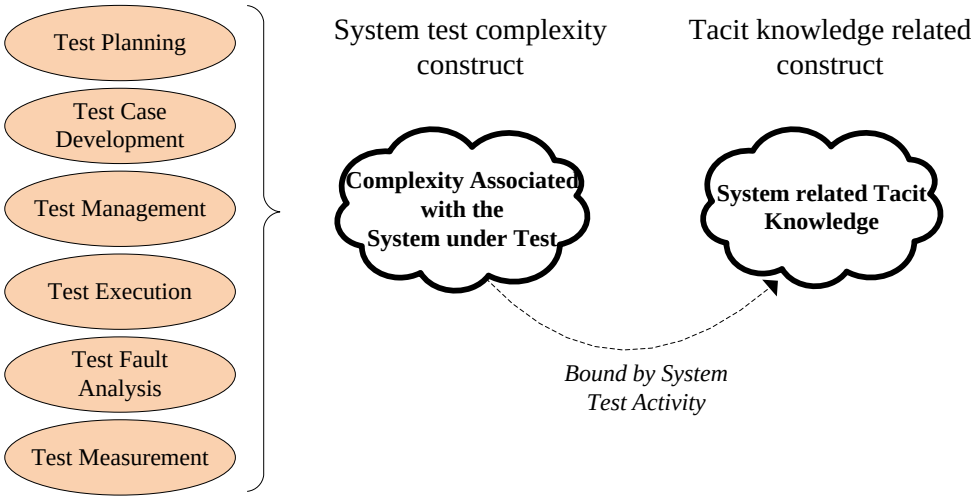


Figure 5.1: Research Model Constructs of the First Hypothesis

Figure 5.1 details the main constructs and indicators associated with this particular hypothesis. The constructs are complexity associated with the system under test, and tacit knowledge associated with the system under test. These constructs are used in conjunction with the following six functions (stages) of system testing, which provides us with indicators for use in the forthcoming coding and analysis:

1. *System test planning*
2. *System test development*
3. *System test execution*
4. *System test fault analysis*
5. *System test measurement*
6. *System test management*

The following section will carry out coding and analysis from a complexity perspective with a subsequent section carrying out the coding and analysis from a tacit knowledge perspective.

5.1.1 Complexity Associated with the System under Test

This section strives to validate the first hypothesis, which proposes a positive relationship between *complexity associated with the system under test and tacit knowledge*. As part of this effort, evidence of complexity associated with the system under test in practice, and associated tacit knowledge, was sought from the collected interview data. Table 5.3 provides a coding and categorisation of data by sentiments expressed. The expressed sentiments have been broken down by system test stage (or function) and by system test activity. A count for the sentiments expressed has been detailed also, with an additional indication as to whether the sentiment is in support of the hypothesis (+) or contrary to the hypothesis (-). Sentiments which add additional information are identifiable by (a).

Test Stage	***System Test Activity***	***Primary Sentiments Expressed***	***Count of similar sentiments***
Test Planning	Understanding features of the system to be tested	Deciding what aspects of the system can and should be tested can be a complex activity.	41 (+)
		This is often due to system interoperability and interdependencies associated with different elements of the system.	17 (a)
		There needs to be a complete understanding of how the feature/system is expected to operate, and how it could be used.	20 (a)
		A lack of understanding at this stage can lead to issues with effective test specification and the estimation of required resources.	6 (a)
Test Development	Test suite development	The implementation of test cases as planned, an activity which must be carried as part of the test development stage, can be quite a complex task.	32 (+)

		This is often due to system interoperability and interdependencies associated with different elements of the system.	8 (a)
Test Execution	Manual test execution	If tests have not been specified properly or clearly defined, then it can introduce complexity at the test execution stage.	12 (+)
		Complexity is more prevalent if testing is manual in nature, as opposed to being automated.	13 (a)
Test Fault Analysis	Debugging potential system issues	System complexity affects the ability to carry our fault analysis or debug on potential issues, and to be able to differentiate between what is an actual bug, and what is a test environment issue. The fault analysis stage demands an understanding of the exact test which was being performed i.e. what the test was attempting to achieve, what effect it had on the system, and what effect it should have had on the system.	33 (+)

Test Measurement	Manual or in-depth analysis of the system under test as part of system quality estimation.	Complexity appears to come into play when deeper analysis is carried out as part of the test measurement stage, in order to accurately evaluate the quality of the system under test.	7 (+)
		A balance must be achieved between adequate system quality against time to market pressures.	2 (a)
Test Management		*Evidence of complexity relating to management of the actual system under test was not found.*	0

Table 5.3: Analysis of Data Relating to Complexity Affecting the System under Test.

As can be seen from table 5.3 evidence involving all stages of system testing, with the exception of *test management*, was identifiable from the interview data. The following section codes and categorises data relating to tacit knowledge relating to the system under test, another important aspect of hypothesis one (outlined at the beginning of this chapter). A model is proposed at the end of 5.1.2 which includes the primary detail from table 5.3.

5.1.2 Tacit Knowledge Associated with the System under Test

The goal of this section is to identify evidence of a positive relationship between activities which have been identified in section 5.1.1 as being impacted by complexity, and tacit knowledge. Tacit knowledge was distinguished from explicit knowledge, through the primary characteristics of being difficult to articulate, and acquired

through experience (in line with the views of Joia and Lemos (2010)). Table 5.4 outlines a coding and categorisation of data by sentiments expressed. The expressed sentiments have been broken down by system test stage (or function) and by system test activity. A count for the sentiments expressed has been detailed also, with an additional indication as to whether the sentiment is in support of the hypothesis (+) or contrary to the hypothesis (-). Sentiments which add additional information are identifiable by an (a).

Test Stage	***System Test Activity***	***Evidence of Tacit Knowledge***	***Count of similar sentiments***
Test planning	Understanding features of the system to be tested.	The availability of tacit knowledge relating to the system under test is essential to enabling effective completion of the planning stage.	41 (+)
Test Development	Test suite development.	The availability of such tacit knowledge relating to the system under test, interoperability etc. is imperative to successfully completing the test development stage. Equally important is knowledge relating to final system deployment.	25 (+)
		For test case development, and to enable effective assessment of automation possibilities, there needs to be an understanding of what has to be tested and how it could be used after	4 (a)

		deployment at a customer site.	
System test execution:	Manual Test Execution	A strong relationship was stated as existing between a manual approach to system testing e.g. load or stress testing, and tacit knowledge.	16 (+)
		A certain amount of the test execution normally lends itself to be made explicit.	21 (-)
System test fault analysis	Debugging potential system issues	To fully appreciate what component of the system bugs are emanating from, one requires tacit knowledge relating to the system under test, specifically relating to how system components interoperate.	24 (+)
		Debugging brings a dependency on development teams for applicable knowledge (or support teams).	17 (a)
		The view was also expressed that a certain amount of debug knowledge can indeed be made explicit.	3 (-)

System test measurement	Manual or in-depth analysis of the system under test as part of system quality estimation.	Required tacit knowledge is associated with current system evaluation against expected, with a balance having to be achieved between available test resources, and the achievement of sufficient quality of the system within a certain timeframe.	9 (+)
		Most of this knowledge lends itself to being made explicit.	25 (-)
		Test measurement lends itself to being automated (and therefore explicit).	6 (-)
System test management		*Any relationship between tacit knowledge associated with test management and tacit knowledge was not evident.*	*0*

Table 5.4: Analysis of Data Relating to Tacit Knowledge Associated with the System under Test.

Figure 5.2: provides an overview of the detail presented in table 5.3 and table 5.4.

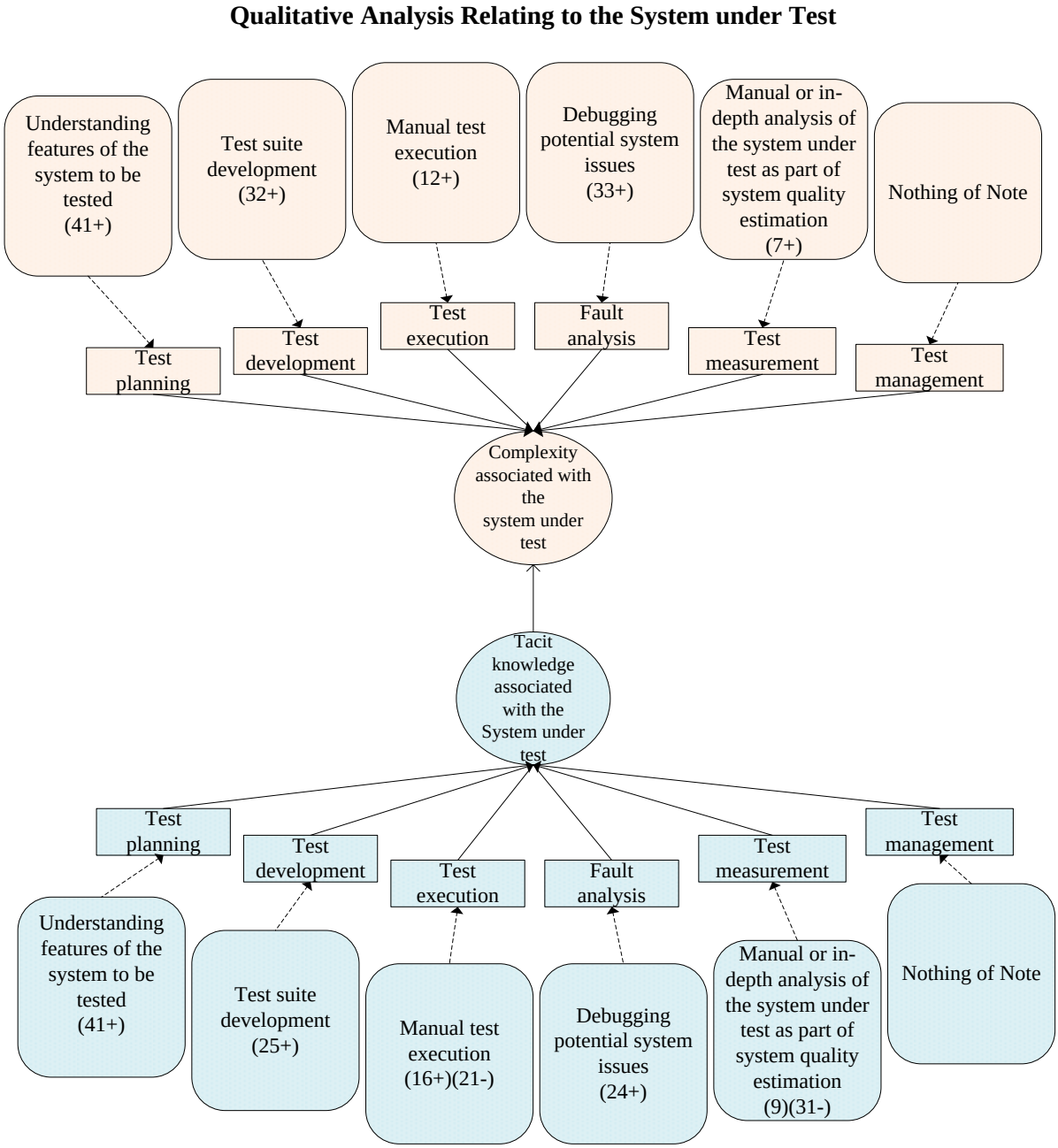


Figure 5.2: Qualitative Analysis Relating to the System under Test.

As can be identified by tables 5.3 and 5.4 and figure 5.2, evidence of complexity and tacit knowledge was found in the case of all stages of system testing stages and activities, with the exception of the *system test management stage*. Similar to tables 5.3 and 5.4, the sentiments expressed in figure 5.2 are accompanied by the count of participants who expressed support for the sentiment (+), and the count of those who contradicted the sentiment.

Figure 5.3 provides the relationships which have been detailed in table 5.4 and figure 5.2, from a socio-technical perspective. The relationships detailed in this table all appear to relate to the *task* of system testing and the system under test (relating to *technology*).

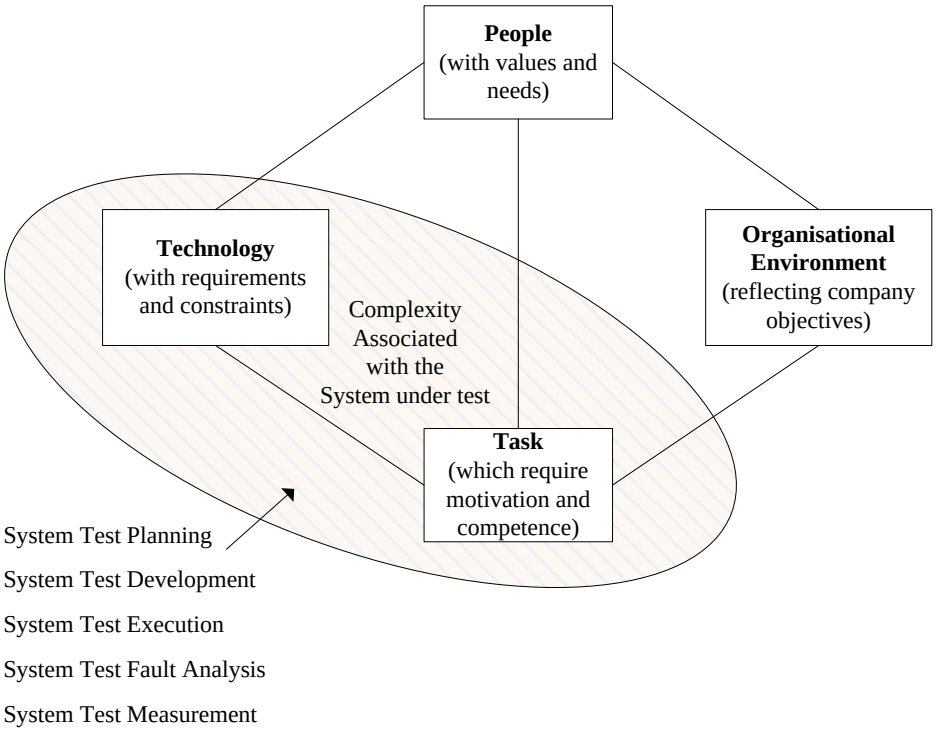


Figure 5.2: Complexity and Tacit Knowledge Associated with the System under Test, from a Socio-Technical Perspective.

The analysis and observations associated with this section are discussed in more detail in the concluding chapter. The following section deals with coding and analysis relating to the second hypothesis, which is concerned with the relationship between complexity associated with the wider process of system testing and tacit knowledge.

5.2 Coding and Analysis of Data Relating to the Second Hypothesis

The second hypothesis proposes that there exists a positive relationship between *complexity associated with the process of system testing* and *tacit knowledge*. This is coded and categorised in sections 5.2.1, and 5.2.2.

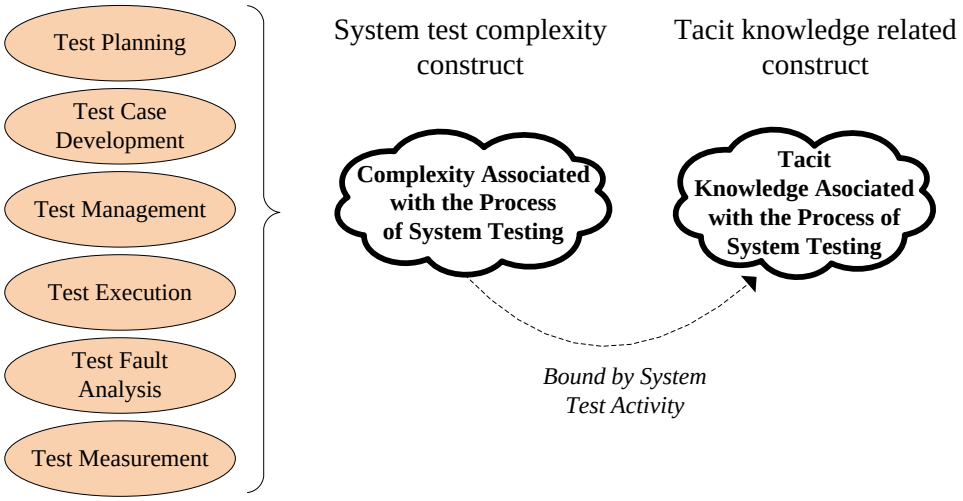


Figure 5.3: Research Model Constructs of the Second Hypothesis.

Figure 5.4 details the main constructs and indicators associated with the this hypothesis. These constructs are complexity associated with the process of system testing, and tacit knowledge associated with the process of system testing. These constructs are used in conjunction with the following six functions (stages) of system testing, which provides us with the indicators for use in the forthcoming coding and analysis:

1. *System test planning*
2. *System test development*
3. *System test execution*
4. *System test fault analysis*
5. *System test measurement*
6. *System test management*

The following section details the coding and analysis from a complexity perspective with the following section dealing with the coding and analysis from a tacit knowledge perspective.

5.2.1 Complexity Associated with the Process of System Testing

This section strives to validate the second hypothesis, which proposes a positive relationship between *complexity associated with the system under test and tacit knowledge*. As part of this effort, evidence of complexity associated with the wider process of system testing, and associated tacit knowledge, was sought from the collected interview data. Table 5.5 provides a coding and categorisation of data by sentiments expressed. The expressed sentiments have been broken down by system test stage (or function) and system test activity. A count for the sentiments has been detailed also, with an additional indication as to whether the sentiment is in support of the hypothesis (+) or contrary to the hypothesis (-). Sentiments which add additional information are identifiable by an (a).

System Test Stage	***System Test Activity***	***Relevant Sentiments Expressed***	***Count of similar sentiments***
System Test Planning	Balancing test resources	Missing or incomplete, functional specifications, relating to system usage can be a contributor to complexity.	15 (+)
		This can influence one's ability to carry out estimation of necessary resources i.e. human, technical and time.	2 (a)

	The selection and prioritisation of test cases	The view was expressed that complexity at the planning stage can affect one's ability to specify appropriate tests, and carry out effective selection and prioritisation of test cases.	16 (+)
		A balance must be achieved between adequate system quality and time to market pressures.	18 (a)
System Test Development	Test environment setup	To build a test environment which is reflective of final deployment can also be a quite complex process.	15 (+)
		There is often a deficit of standards or guidance to enable test environments to accurately reflect those of customers. There is often insufficient knowledge relating to the actual deployed system in practice.	20 (a)
		There can be multiple different routes for successful testing to be achieved and this introduces a certain amount of complexity.	7 (+)

	Accommodating test automation.	The role of test automation was also cited as a potential contributing factor to complexity i.e. what should be automated and how?	11 (+)
System test execution:	Manual test execution with incomplete test case specifications.	If tests have not been specified properly or clearly defined, then it can introduce complexity at the test execution stage.	17 (+)
		This has been described as being particularly relevant if testing is manual in nature e.g. exploratory or non-standard testing, as opposed to being automated.	13 (+)
		A lot of this knowledge can be made explicit.	12 (-)
System test fault analysis	Debugging potential test environment issues.	The effects of complexity are often visible at the fault analysis stage, when you must determine is a failure due to automation or due to actual system failure.	19 (+)

		The automation of test cases is described as something which contributes greatly to general complexity associated with system testing. Sometimes automation masks the exact system interoperability, thereby having the effect of reducing the general understanding of system operation.	4 (+)
System test measurement	Development, execution, or interpretation of manual or in-depth quality analysis.	Complexity appears to come into play when an automated test measurement framework has not been implemented, or when deeper analysis is carried out, in order to accurately evaluate the quality of the system under test.	14 (+)
		Balancing system quality and time to market pressures can also prove a complex activity.	3 (+)
		System test measurement does lend itself to being made explicit and automated, particularly if kept simplistic (pass or fail).	21 (-)

System test management	Management of resources.	Management of resources, which involves the balancing of resources associated with the test environment, enabling test case preservation, can be quite a complex task. Such management is stated as requiring experience and know-how in order to balance resources properly.	22 (+)
		Most of this knowledge can be made explicit.	4 (-)

Table 5.5: Analysis of Data Relating to Complexity Associated with the Process of System Testing.

The following sections carry out further analysis on the concept of tacit knowledge as it relates to the wider system test process. This is an important aspect of the second hypothesis, referred to at the beginning of section 5.1. A model is proposed at the end of 5.2.2 which includes the primary detail from table 5.5.

5.2.2 Tacit Knowledge Associated with the Process of System Testing

The goal of this section is to identify evidence of a positive relationship between activities which have been identified in section 5.2.1 as being impacted by complexity, and tacit knowledge. Similar to the previous hypothesis, tacit knowledge was distinguished from explicit knowledge, through the primary characteristics of being difficult to articulate, and acquired through experience. Table 5.6 provides a coding and categorisation of data by sentiments expressed. The expressed sentiments have been broken down by system test stage (or function) and by system test activity. A count for the sentiments expressed has been detailed also, with an additional indication

as to whether the sentiment is in support of the hypothesis (+) or contrary to the hypothesis (-). Sentiments which add additional information are identifiable by an (a).

System Test Stage	***System Test Activity***	***Relevant Sentiments Expressed***	***Count of similar sentiments***
Tacit knowledge relating to the task of system test planning	Balancing of test resources.	The importance of tacit knowledge relating to test case planning, which is gained through experience, has been emphasised by numerous participants.	27 (+)
	Prioritisation and selection of test cases.	A shortfall in tacit knowledge could result in a lack of appreciation for what tests are necessary in order to test the system properly, given available resources.	7 (+)
		A certain amount of knowledge relating to planning does lend itself to being made explicit e.g. via specifications etc.	9 (-)
Tacit knowledge relating to the task of system test development	Test environment setup	Applicable test environment development knowledge is usually tacit in nature and difficult to make explicit.	28 (+)

		Knowledge relating to the test environment may not be as easy to acquire if the system being implemented is a bespoke system, being developed from scratch by a separate team e.g. automation team, or in the case of a geographically dispersed test team.	5 (a)
		A contrary view was expressed by a minority that a lot of test environment knowledge can be made explicit.	3 (-)
Tacit knowledge relating to the task of system test execution	Manual test execution with incomplete test case specifications.	The views were expressed that tacit knowledge is often involved when a manual approach to testing is taken. This may involve complex test steps, and may form part of load testing or exploratory testing, which would require more detailed test environment knowledge.	16 (+)
		A certain amount of the test execution knowledge normally lends itself to be made explicit.	21 (-)
Tacit knowledge relating to the task of fault analysis	Debugging potential test environment issues.	An ability to debug is primarily dependent on the experience and tacit knowledge of the tester.	16 (+)

		When carrying out fault analysis, one needs to rule out the involvement of the test environment, as opposed to the system under test.	19 (a)
		Knowledge associated with automated test environments, is described as often being primarily tacit in nature. Debugging of issues associated with automated environments, often brings a dependency on other team members (including those focussed on development and maintenance of the test environment)	6 (+)
Tacit knowledge relating to the task of system test measurement	Development, execution, or interpretation of manual or in-depth quality analysis.	Required tacit knowledge is associated with system evaluation, and achieving a balance between resources, and the achievement of sufficient level of system quality within a defined timeframe.	9 (+)
		Test case measurement is described as being based on experience, but something with a weak relationship to tacit knowledge.	25 (-)

		Test case measurement can be taken care of, to a large extent, on an automated basis (by its nature explicit), which simplifies matters.	6 (-)
Tacit knowledge relating to the task of system test management	Tacit knowledge relating to the task of system test management	The dependence on tacit knowledge appears to be required with the introduction of new systems, modifications to test environments, or optimisation efforts, all of which can also make test environments quite complex to manage.	7 (+)
		Most of test case management does lend itself to being made explicit.	20 (-)

Table 5.6: Analysis of Data Relating to Tacit Knowledge Associated with the System under Test.

Figure 5.5: provides an overview of the detail presented in tables 5.5 and 5.6.

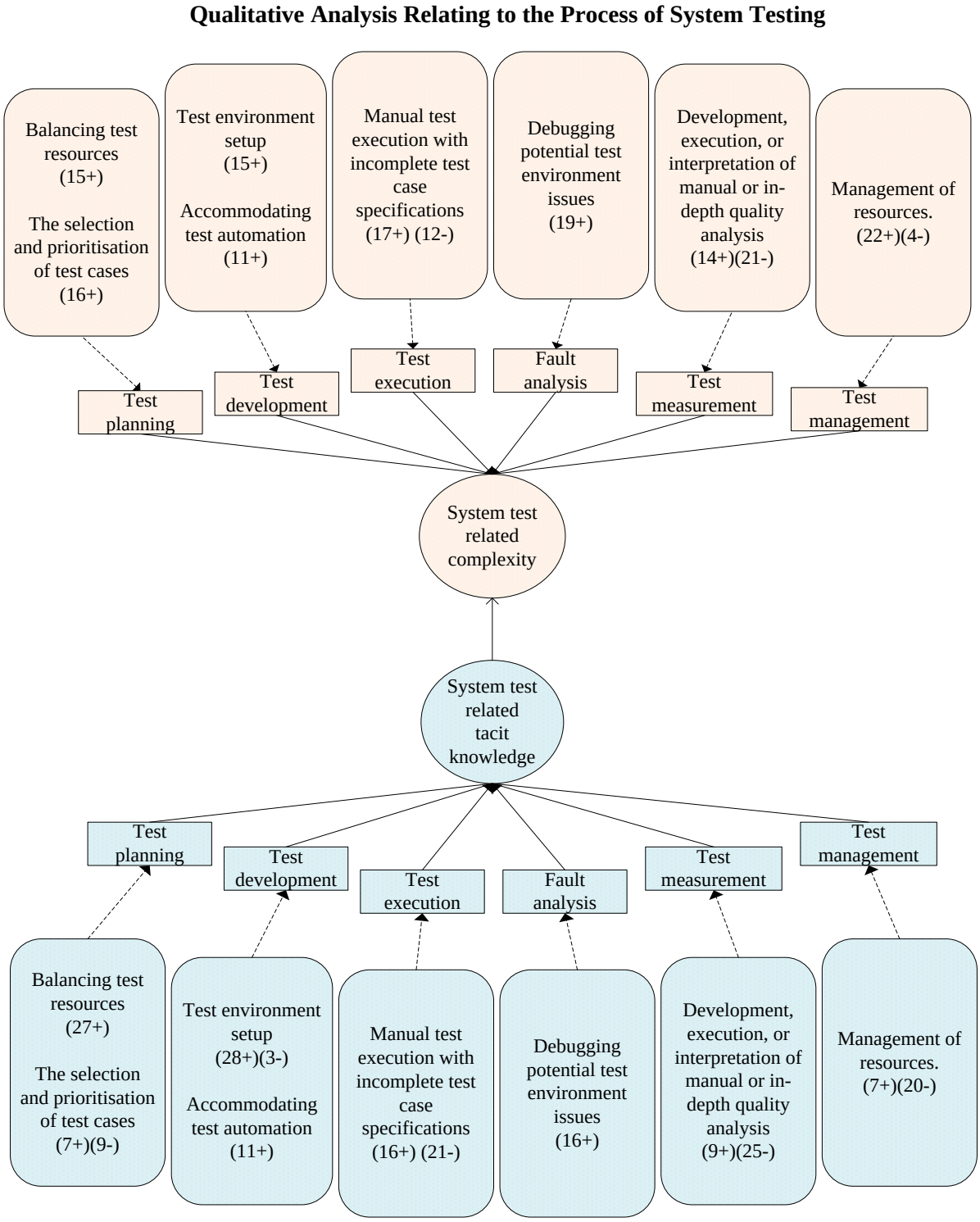


Figure 5.4: Qualitative Analysis Relating to the System under Test.

Similar to table 5.6, the sentiments expressed in figure 5.5 are accompanied by the count of participants who expressed support for the sentiment (+), and the count of those who contradicted the expressed sentiment.

Figure 5.6 provides us with an overview of the evidence detailed in table 5.6, from a socio-technical perspective.

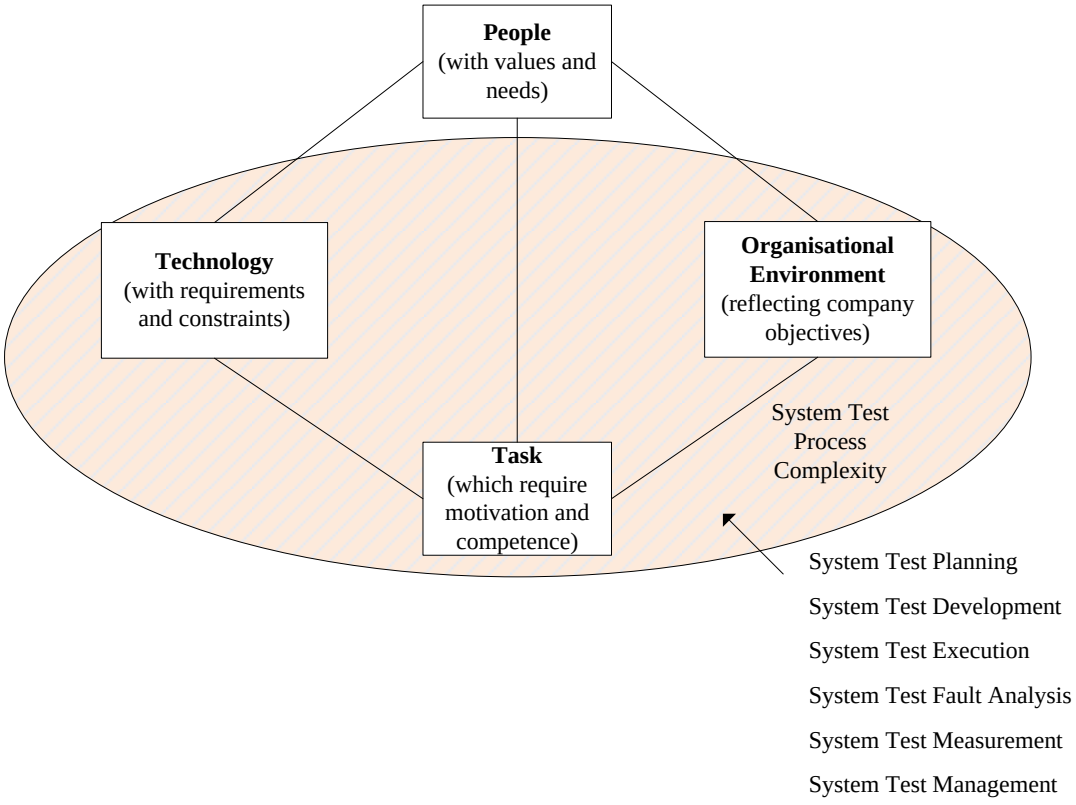


Figure 5.5: Complexity and Tacit Knowledge Associated with the Process of System Testing, from a Socio-Technical Perspective.

In this particular case, there appears to be a greater influence of *organisation* or project drivers or complexity, along with complexity associated with the *task* related complexity.

The following section provides us with an overview of the quantitative analysis which has been conducted. A synopsis of the qualitative analysis which has been conducted in sections 5.1.1, 5.1.2, 5.2.1, and 5.2.2, is carried out as part of the concluding section of this chapter i.e. section 5.5.

5.3 Analysis of Quantitative Data

As previously highlighted, there are two primary constructs in the proposed research model, relating to *system test related complexity*, and *system test related tacit knowledge*. These constructs are used in conjunction with the following six functions (stages) of system testing, to provide us with indicators for use in the following sections:

1. *System test planning*
2. *System test development*
3. *System test execution*
4. *System test fault analysis*
5. *System test measurement*
6. *System test management*

The indicators used are formative in nature, and cover the entire domain space of the system test complexity construct, as recommended by Kim et al. (2010). These indicators are also in line with the stages of system testing as outlined by Eickelmann and Richardson (1996), and Desai and Shah (2011). The bases for the quantitative analysis are responses provided to questions 3, 4, and 5, as outlined in table 4.1. The data in table 5.7 provides a synopsis of the relationships between complexity associated with the different stages of system testing, and tacit knowledge. Figures detailed, are derived from data displayed in figure 5.7. Confidence levels are detailed in brackets:

1. System test measurement showed a strong relationship to complexity. Test case planning and test case management also displayed a reasonably strong relationship to complexity, with ~76%, and 72% level of confidence, respectively. The other stages, system test development, system test execution and system test fault analysis, all displayed a relatively weak relationship to complexity.
2. System test measurement again showed a strong relationship to tacit knowledge (>99% level of confidence). Besides system test management (~68% level of confidence), system test development (~54% level of confidence), and system test fault analysis (~50% level of confidence) displayed reasonable relationship to

system test related tacit knowledge, with system test planning (~12% level of confidence), and system test execution (~27% level of confidence), displaying rather weak relationships.

Stage	***Relationship to System Test Complexity***	***Relationship to Tacit Knowledge***
System Test Planning	(~76%)	(~12%)
System Test Development	(~27%)	(~54%)
System test Execution	(~41%)	(~27%)
System Test Fault Analysis	(~13%)	(~50%)
System Test Measurement	(>99%)	(>99%)
System Test Management	(~72%)	(>68%)

Table 5.7: Quantitative Evidence of System Test Complexity and Tacit Knowledge.

In addition to the aforementioned, the following relationships were also identifiable:

The bivariate correlation values highlight significant relationships ($p < 0.05$) between the following system test complexity indicators:

- *System test planning* and both *system test development*, *system test execution*, and *system test management*.
- *System test development* and both *system test fault analysis* and *system test management*.
- *System test execution* and *system test measurement*, and *system test management*.
- *System test fault analysis* and both *system test measurement.*

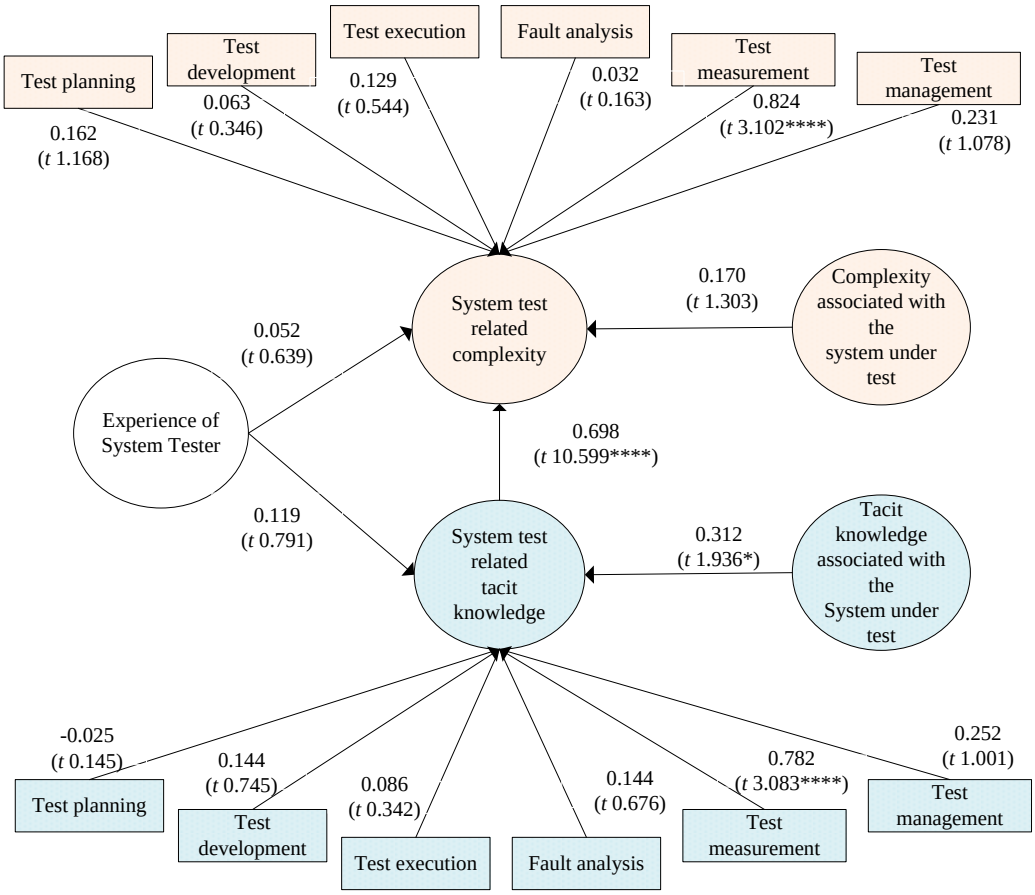
The bivariate correlation values highlight significant relationships ($p<0.05$) between:

- *System test planning* and both *system test execution* and *system test measurement*.
- *System test development and fault analysis.*
- *System test fault analysis* and *system test management*.
- *System test measurement* and *system test management*.

Further analysis was carried out using a partial least squares (PLS) approach. This is discussed in detail in the following section.

5.3.1 Modelling the Quantitative Data

The data was analysed using a variance based, partial least squares (PLS), structure equation modelling (SEM) approach. This was primarily chosen because of its recommended use in the case of formative variables, but also because of the recognised benefit of such an approach in theory development (Hair, Ringle, & Starstedt, 2011), and in accommodating smaller sample sizes, of between 50 and 100 participants (Iacobucci, 2010). The principal PLS-SEM tool used was SmartPLS. Further validation of the bivariate correlation between indicators was carried out adopting a covariance based approach, using IBM SPSS. Taking into account the guidance, as provided by Cenfetelli and Bassellier (2009), figure 5.7 shows the indicator weightings and *t*-values associated with the *system test related complexity* construct.



*NB: *p<0.10; **p<0.05;* **p<0.01;****p<0.0005;*

Figure 5.6: Model of Quantitative Results.

There was no evidence of any multicollinearity in effect because, as detailed, the VIF values detailed in figure 5.8, are all well below a generally recommended rule of thumb of being less than a value of 5.0 (Hair, Ringle, & Starstedt, 2011), 4.0 (O'Brien, 2007), and 3.3 ((Diamantopoulos & Siguaw, 2006), (Marciniak, Amrani, Rowe, & Adam, 2014)). In addition to the weightings, and t-statistic values, table 5.8 provides us with the variance inflation factors associated with the system test complexity indicators.

System Test Complexity Indicators	**Weights**	***t*-Values**	**Variance Inflation Factor**
System test planning (SC1)	0.162	1.168	1.246
System test development (SC2)	0.063	0.346	1.260

System test execution (SC3)	0.129	0.544	1.210
System test fault analysis(SC4)	0.032	0.163	1.171
System test measurement (SC5)	0.824	3.102****	1.190
System test management (SC6)	0.231	1.078	1.176

*NB: *p<0.10; **p<0.05;* **p<0.01;* ***p<0.0005;*

Table 5.8: System Test Complexity Indicators.

As can be seen from the results, only the weighting associated with *system test measurement*, is shown as significant (having a t-statistic equating to 3.102, which is representative of a greater than 99% level of confidence). It must also be noted, that whilst the other weightings may not be highly significant, the values associated with *system test planning* and *system test management*, against *system test related complexity*, are not insignificant, equating to ~76%, and ~72%, levels of confidence, respectively.

Table 5.9 provides us with the bivariate correlations between the indicators detailed in figure 5.7.

	SC1	***SC2***	***SC3***	***SC4***	***SC5***	***SC6***
SC1	1.000					
SC2	0.334***	1.000				
SC3	0.309***	0.172	1.000			
SC4	0.061	0.270**	0.119	1.000		
SC5	0.016	-0.025	0.259**	0.248**	1.000	
SC6	0.238**	0.241**	0.222**	0.194*	0.221	1.000

*NB: *p<0.10; **p<0.05;***p<0.01;*

Table 5.9: Bivariate Correlations between System Test Complexity Indicators.

The bivariate correlation values highlight significant relationships ($p<0.05$) between the following system test complexity indicators:

- *System test planning* and both *system test development*, *system test execution,* and *system test management*.
- *System test development* and both *system test fault analysis* and *system test management*.
- *System test execution* and *system test measurement*, and *system test management*.
- *System test fault analysis* and both *system test measurement.*

Similar to table 5.8, table 5.10 provides us with the results of initial analysis of the indicators, but in this case relating to the other primary construct, *system test related tacit knowledge.*

System Test Tacit Knowledge Indicators	**Weights**	**t-value**	**Variance Inflation Factor**
System test planning (ST1)	-0.025	0.145	1.357
System test development (ST2)	0.144	0.745	1.078
System test execution (ST3)	0.086	0.342	1.282
System test fault analysis(ST4)	0.144	0.676	1.095
System test measurement (ST5)	0.782	3.083****	1.235
System test management (ST6)	0.252	1.001	1.337
*NB: *p<0.10; **p<0.05;***p<0.01;****p<0.0005;*			

Table 5.10: System Test Tacit Knowledge Indicators.

Similar to the system test complexity indicators, table 5.10 also highlights only the values associated with *system test measurement*, as being significant (having a t-statistic or t-value of 3.083, equating to a level of confidence greater than 99%). The value associated with *test management* displays a confidence level close to equating to ~70%. Both system test development and system test fault analysis, display moderate levels of confidence of ~50%. The VIF values detailed in table 5.10 do not show any

evidence of excessive multicollinearity, again being less than a generally recommended rule of thumb of being less than a value of 5.0 (Hair, Ringle, & Starstedt, 2011), 4.0 (O'Brien, 2007), 3.3 ((Diamantopoulos & Siguaw, 2006), (Marciniak, Amrani, Rowe, & Adam, 2014)).

Table 5.11 details the bivariate correlations between the indicators detailed in figure 5.7.

	ST1	***ST2***	***ST3***	***ST4***	***ST5***	***ST6***
ST1	1.000					
ST2	0.097	1.000				
ST3	0.457***	0.135	1.000			
ST4	0.124	0.457***	0.135	1.000		
ST5	0.272**	0.138	0.155	0.141	1.000	
ST6	0.239*	0.247*	0.165	0.284**	0.393***	1.000

*NB: *p<0.10; **p<0.05;***p<0.01;*

Table 5.11: Bivariate Correlations between System Test Tacit Knowledge Indicators.

The bivariate correlation values highlight significant relationships ($p<0.05$) between:

- *System test planning* and both *system test execution* and *system test measurement*.
- *System test development and fault analysis.*
- *System test fault analysis* and *system test management*.
- *System test measurement* and *system test management*.

The next section performs a comparison of the qualitative data which has been previously covered, and the quantitative data covered in this section.

The following section provides a brief overview of actions which can be taken to combat the effects of system test complexity.

5.3.2 A Comparison between the Qualitative and Quantitative Analysis

Table 5.12 highlights those relationships with a quantitative rating equivalent to greater than 70% level of confidence.

Stage	***Relationship to System Test Complexity***	***Relationship to Tacit Knowledge***
System Test Planning	(quantitative)	
System Test Development		
System test Execution		
System Test Fault Analysis		
System Test Measurement	(quantitative)	(quantitative)
System Test Management	(quantitative)	

Table 5.12: Discussion of Quantitative Results.

When compared to the qualitative analysis, the following discrepancies are obvious regarding which activities displayed a relationship between complexity and tacit knowledge:

1. The quantitative data did not appear to highlight a relationship to complexity at the *test planning* stage. This is at odds with the previously discussed qualitative data which highlighted strong support, amongst participants, in line with the following sentiment which was regularly expressed: *with testing of complex systems, the availability of tacit knowledge relating to the system under test, interoperability etc. is imperative to enable effective completion of the planning and test development stages*.

2. Quantitative data relating to the *test development* stage appears to be also at odds with the qualitative data, which appears to show a general consensus amongst testers that *there is often a deficit of standards or guidance regarding the setup of test environments which accurately reflect customer deployments, and often insufficient knowledge relating to the actual system in practice*. This was generally stated as having particular relevance to the task of system test development. The availability of separate independent test environment support teams, and the assistance of more experienced team members, may help explain why some participants did not perceive there to be high levels of complexity associated with this stage, thus explaining the variance in reported values. Both the availability of a separate test development team, and the availability of more experienced team members, was referred to as helping to reduce complexity associated with the system test development stage.
3. A strong positive relationship to complexity or tacit knowledge associated with *test execution* does not appear to be acknowledged from a quantitative perspective. The quantitative analysis does not appear to be keeping with the commonly expressed sentiment that *if tests have not been specified properly, or clearly, then it can introduce complexity at the test execution stage*. Having said that, numerous experienced participants went on to state that *a lot of this knowledge can be made explicit*, with little support being displayed for a strong relationship between system test execution and tacit knowledge. The qualitative data did show support for complexity associated with the system under test, and the wider system test process, when a manual approach to test execution is employed, as opposed to use of an automated infrastructure.
4. Quantitative data displayed a weak positive relationship between the *fault analysis* stage of testing, to both complexity and tacit knowledge. This in contrast to the qualitative data which appeared to highlight a positive relationship to both complexity and tacit knowledge. A strong relationship to development teams as a source of tacit knowledge applicable to this particular stage was also highlighted as part of the qualitative data.
5. The quantitative data displayed a positive relationship between system test measurement and tacit knowledge. The qualitative data displayed a similar relationship, but associated with a manual test measurement approach, and also

relating to the achievement of a balance between quality and time to market pressures.

6. There was perceived to be a positive relationship between system test management and complexity, from analysis of quantitative data. The significance of complexity to the management stage was not apparent from the qualitative data, from a system under test perspective, but there was evidence of complexity from a wider system test process perspective. This related to a manual approach to test environment management being adopted.

The lower quantitative results can be explained in most case by the high median but high variance between ratings which were provided. One explanation for this variance between ratings could be the employment of positive actions which are actively being taken by some test teams, with the purpose of complexity reduction. This would explain the lower complexity and tacit knowledge ratings in those particular cases. Another point to consider is the possible lack of consistent appreciation and recognition for the presence and effect of tacit knowledge amongst participants. These reasons might go some way towards explaining the inconsistent ratings for complexity and tacit knowledge, versus the qualitative analysis linked to a series of open questions, which reflected a stronger presence of complexity and tacit knowledge for the various stages of system testing. The following section continues support for the qualitative analysis, highlighting the research findings from a socio-technical perspective.

The following section provides an overview of recommended actions which could be taken as part of efforts to reduce the effects of complexity.

5.4 Identified Actions for Dealing with System Test Complexity

As an outcome of the interview stage, a number of actions were identified as having a positive effect in the reduction of complexity associated with system testing. The support of system testing and facilitating the flow of knowledge, have been identified as being of the upmost importance. The following key areas were identifiable:

1. *The availability of knowledge within the test team.*
2. *The availability of knowledge from development teams.*
3. *The use of support applications and support teams*

Table 5.13 provides a coding and categorisation of the three aforementioned areas, by sentiments expressed during the interview stage. The expressed sentiments have been broken down by the knowledge source and system test activity.

Action	***Knowledge Source***	***Evidence of Tacit Knowledge***	***Count of similar sentiments***
The availability of knowledge within the test team.	*The dependence on tacit knowledge from team members and the availability of SMEs.*	It has been explained that the availability of subject matter experts, providing necessary tacit knowledge relating to the actual system under test and the system test environment, is important in the reduction of complexity.	43
	The importance of explicit knowledge in reducing complexity	At planning stages, there is a great deal of information which can be made explicit via function specifications, user stories etc. which can help in reducing complexity associated with system testing.	46
The availability of	*The importance of the transfer of*	Due to the inadequacies of formal documentation, a	25

knowledge from development teams	*tacit knowledge from developers in reducing system test complexity*	significant amount of time is spent trying to acquire tacit knowledge from development teams, especially in relation to system interactions and expected outcomes under different operating conditions.	
The benefit of support applications and support teams	*The use of development and test support teams and support applications.*	There is a benefit of providing test support teams, which are separate to system testing, but closely aligned, such as project management teams or test environment support teams e.g. test environment automation teams. Such teams provide ongoing support for system testing from a development process and a test environment perspective.	14
		The use of automated systems which may be custom built or off the shelf, can help significantly in reducing complexity associated with test case execution and measurement stages of system testing.	5

	The use of project management applications	Project management tools such as JIRA and Confluence were described as helping to clarify what architectural decisions have been made, and the principal drivers. Tools such as wiki pages are described as being effective to detail such architectural decisions.	2

Table 5.13: A Breakdown of Actions which may be taken to reduce the Effects of Complexity.

Figure 5.8 provides us with an overview of this section, from a socio-technical perspective. Included are the main actions areas which have been identified, i.e. *test team knowledge*, *development team knowledge*, *support applications*, and *support teams*. Interestingly, the identified actions against complexity, relate to interactions with development, test, automation, and project management teams i.e. *people* interactions, and *technological* solutions, such as project management and automation applications.

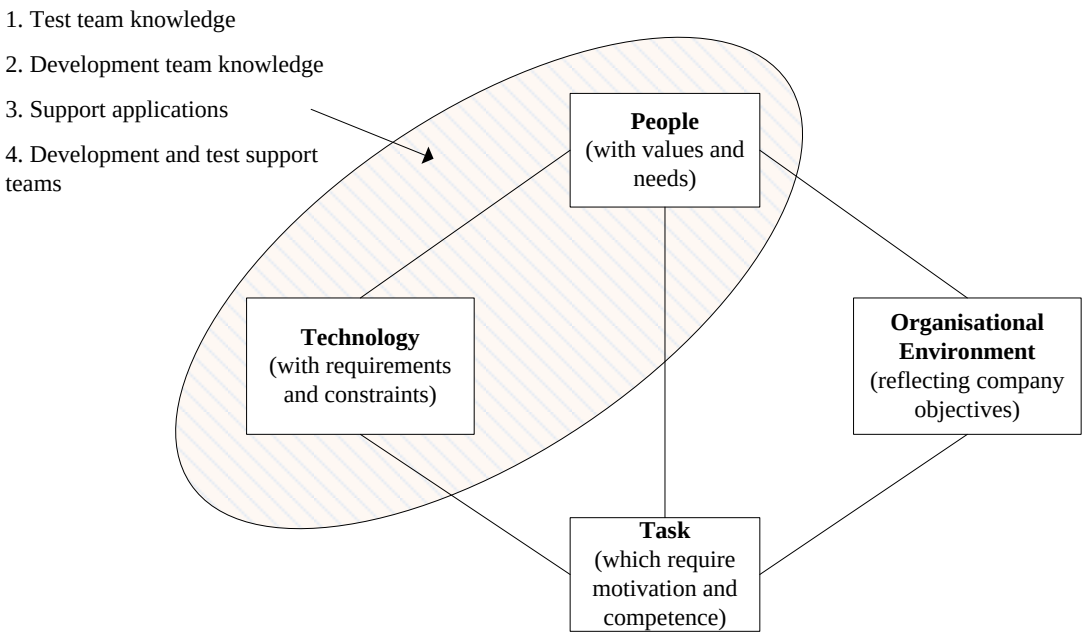


Figure 5.7: Actions which may be taken to reduce the Effects of Complexity, from a Socio-Technical Perspective.

5.5 Modelling Research Findings

This section provides a model of the analysis which has been conducted as part of sections 5.1, 5.2, and 5.4. Activities with a common positive relationship to system test complexity and tacit knowledge have been identified in sections 5.1 and 5.2. Section 5.1 highlighted system test activities which are affected by complexity associated with the system under test which have a positive relationship to tacit knowledge. Section 5.2 identified activities which are primarily affected by complexity associated with the wider process of system testing, and which also have a positive relationship to tacit knowledge. The following section 5.5.1, models observations from the coding and categorisation which has taken place in sections 5.1 and 5.2. Section 5.5.2 models the highlighted actions which have been taken to reduce the effects of system test complexity, as detailed in section 5.4. The final section of this chapter provides a socio-technical representation of the research findings.

5.5.1 A Model of the Relationship between Complexity and Tacit Knowledge

Figure 5.9 highlights those activities (detailed in section 5.1 and section 5.2), which are affected by complexity (as provided in the previous chapter), and display a positive relationship to tacit knowledge. The model details activities from both a *system under test* and a *wider system test process* perspective.

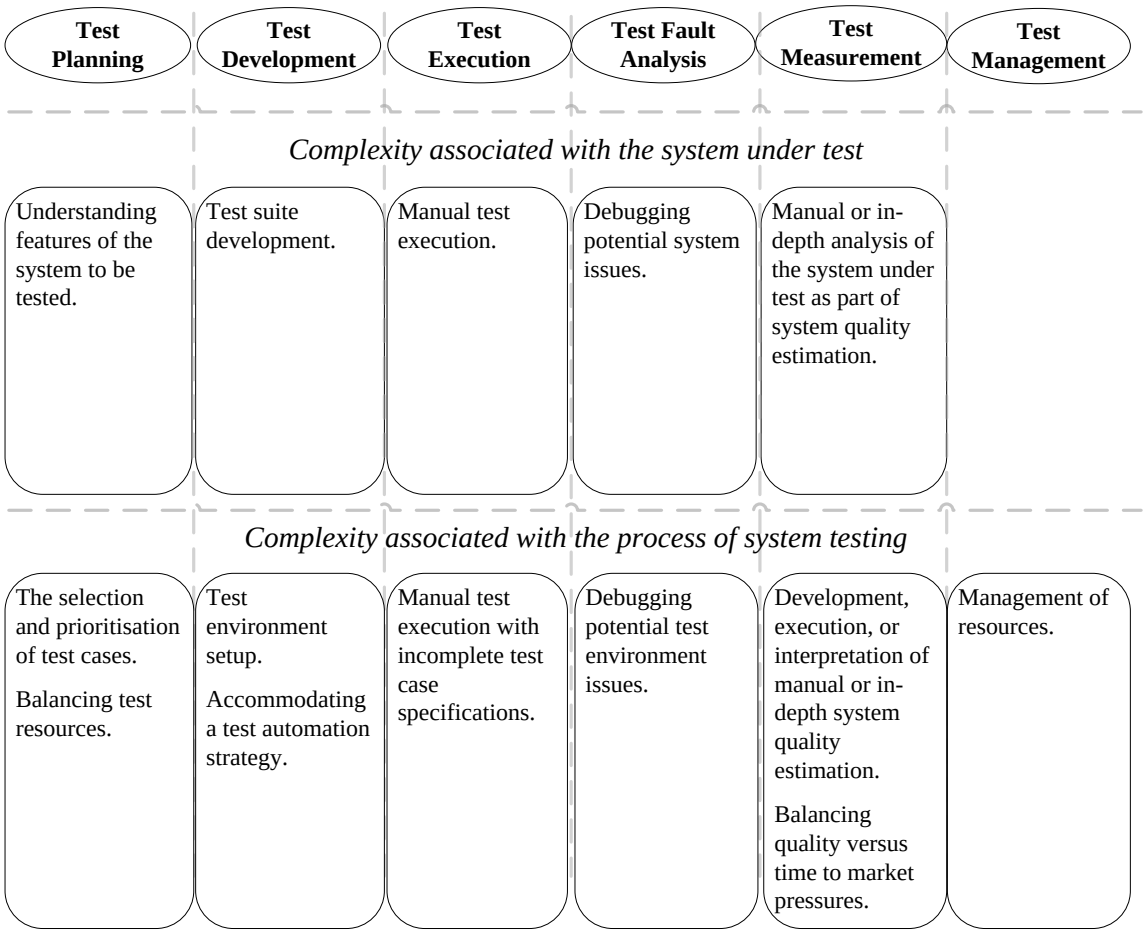


Figure 5.8: A Model of Sources of Complexity with a Direct Relationship to Tacit Knowledge.

Complexity associated with the system under test was perceived to be very important in the case of a number of activities during the test process. Prior to test execution, activities such as *understanding the system features to be tested* (required for test planning), *development of individual test suites* (test cases), and *manual test execution* (as opposed to automated test execution), all appear to be relevant. Complexity

associated with *manual test execution* was linked to the correct execution of tests against the system under test, as opposed to complexity related to the test environment. After test execution, activities such as *debugging of potential issues* from a system under test perspective, and manual efforts to estimate system quality, were also found to have a relationship to both complexity and tacit knowledge.

Complexity was also found, and detailed in figure 5.9, relating to activities associated with the wider system test process. Prior to test execution, *the prioritisation and selection of test cases*, and *balancing test resources*, have been found to be potentially complex at the test planning stage (the criteria for selection of test cases has been referred to as including coverage criteria, resource constraints, and fault detection capability (Lin, Chou, Lai, Huang, & Chung, 2012)). At the test development stage evidence of complexity associated with *setup of the test environment*, and *accommodation of an automated test strategy*, was also found to be potentially complex. *Manual test execution* was found to be complex from a test environment perspective, particularly when tests have not been specified properly. After test execution, complexity can affect activities such as *debugging potential test environment issues* as part of the fault analysis stage. Complexity can also be associated with activities associated with a *test measurement framework*, independent of the system under test, and also *achieving a balance between quality and time to market pressures*. Test management can be affected by complexity associated with the *management of test resources*, whereby the test environment must be preserved, with a view to ensuring consistent test repeatability, which can be difficult to achieve if the test environment is not being used exclusively but is rather being shared amongst different teams.

In addition to the identification of complexity with a relationship to tacit knowledge, the previous chapter also identified actions which were suggested as having a positive effect in the reduction of complexity associated with system testing. These actions are discussed in more detail in the next section.

5.5.2 A Model of Proposed Actions to Reduce the Effects of Complexity

Actions have been identified as part of section 5.4 which encourage the availability of both tacit and explicit through knowledge transfer, and also encourage the conversion of appropriate tacit knowledge to explicit of knowledge. Three primary sources of knowledge which have been identified are:

- *The availability of knowledge within the test team*, both from a personal and team perspective. Such knowledge, as often held by subject matter experts (SMEs), can have a positive effect on the reduction of complexity associated with test planning, test development, test case execution, and test fault analysis.
- *The availability of accessible tacit knowledge from development teams* can have a positive effect on the reduction of complexity associated with the test planning, test execution, and test fault analysis stages.
- *The use of support applications and support teams* has been highlighted as being beneficial in the reduction of system test complexity associated with the test planning, test development, test execution, test management and test measurement stages.

A model of the identified actions is detailed in figure 5.10.

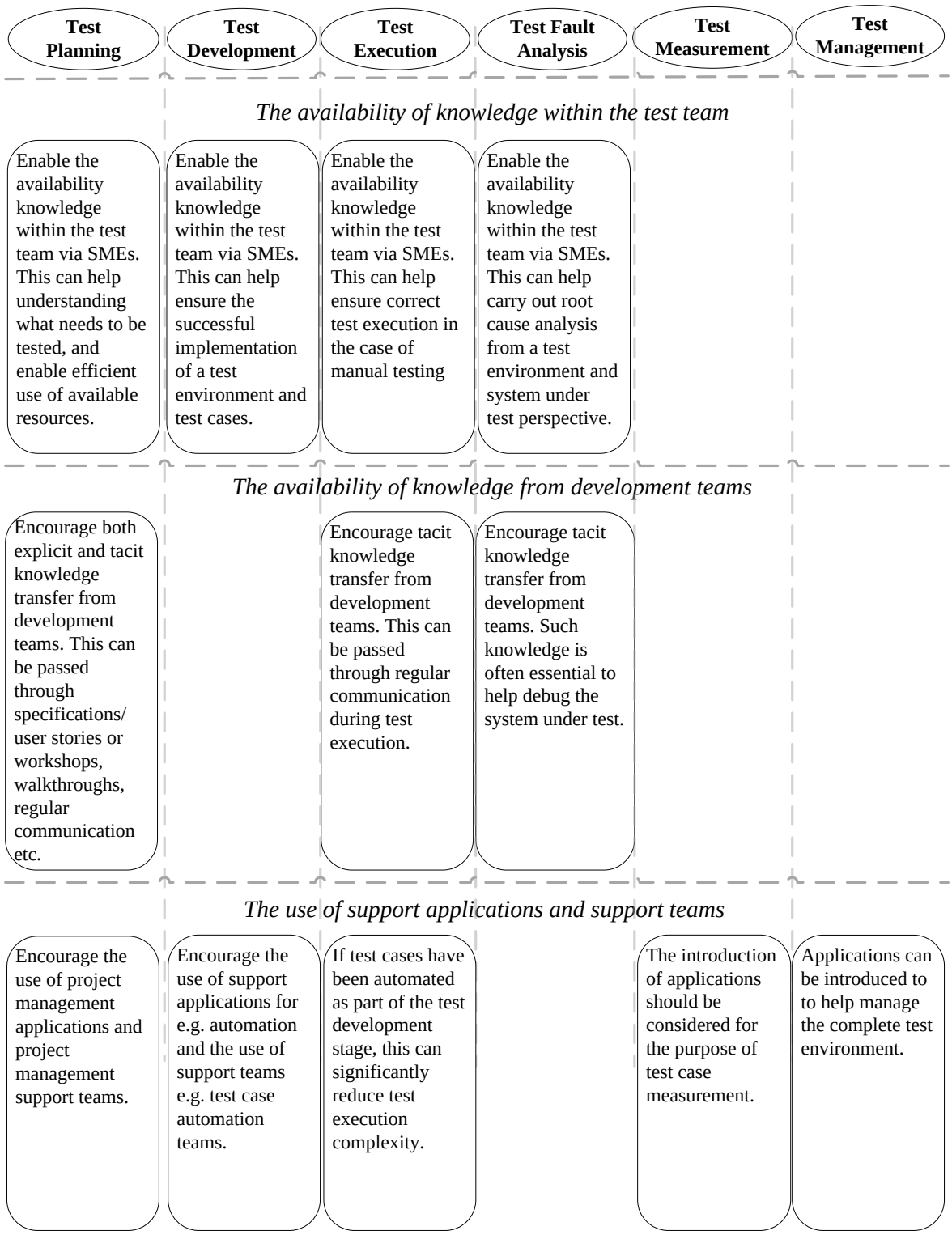


Figure 5.9: A Model of Recommended Actions to Reduce the Effects of Complexity.

The availability of test team knowledge via subject matter experts (SMEs) was found to provide benefit at the test case planning (providing knowledge relating to system

understanding and resource management), test case development (test environment development and the development of test cases), test execution, and fault analysis stages. Complexity can be reduced at the test development and test execution stages by the introduction of an automated test environment, with a separate team tasked with handling the setup of such an environment.

Regarding the availability of tacit knowledge from development teams, the availability of both explicit and tacit knowledge from developers has been shown to be important at the test planning, test execution, and fault analysis stages, but of lesser importance at the test development, test measurement, and test management stages. This could be explained by a reliance on development teams for initial system understanding, but a diminished reliance at the test development stage, because of previously acquired knowledge at the test planning stage. Validation of the test environment, from a development perspective, can come as part of the test execution stages, and test fault analysis stages. Outside of the transfer of tacit knowledge from developers, the transfer of knowledge which can be made explicit relating to the system under test has also been shown to be important. Such knowledge is usually passed via specifications, user stories etc. The conversion to explicit knowledge was also evident through comments referring to the benefit of the use of support applications, and test measurement applications, which is effectively making explicit, knowledge relating to those particular aspects of system testing.

The use of support applications and support teams has been found to be beneficial at all stages with the exception of the fault analysis stage.

5.5.3 The Identified Research Findings from a Socio-Technical Perspective

This section details the qualitative research findings from a socio-technical perspective. As explained in the previous section, the disparity between quantitative and qualitative analysis was attributed to the high median, but high variance, between ratings which were provided as an indication of complexity and tacit knowledge. It is argued that the qualitative analysis provided a greater insight into the relationship to system test complexity, and tacit knowledge, due to the use of open questions, a

technique which has previously been used to good effect by Kaplan and Duchon (1988), and Kothari et al. (2012). There is recognised benefit in applying a socio-technical model to the research findings ((Herbsleb, 2007), (Lu, Xiang, & Wang, 2011), (Sommerville, et al., 2012), (Davis, Challenger, Jayewardene, & Clegg, 2013)). Figure 5.11 highlights the output from the qualitative analysis, from a socio-technical perspective.

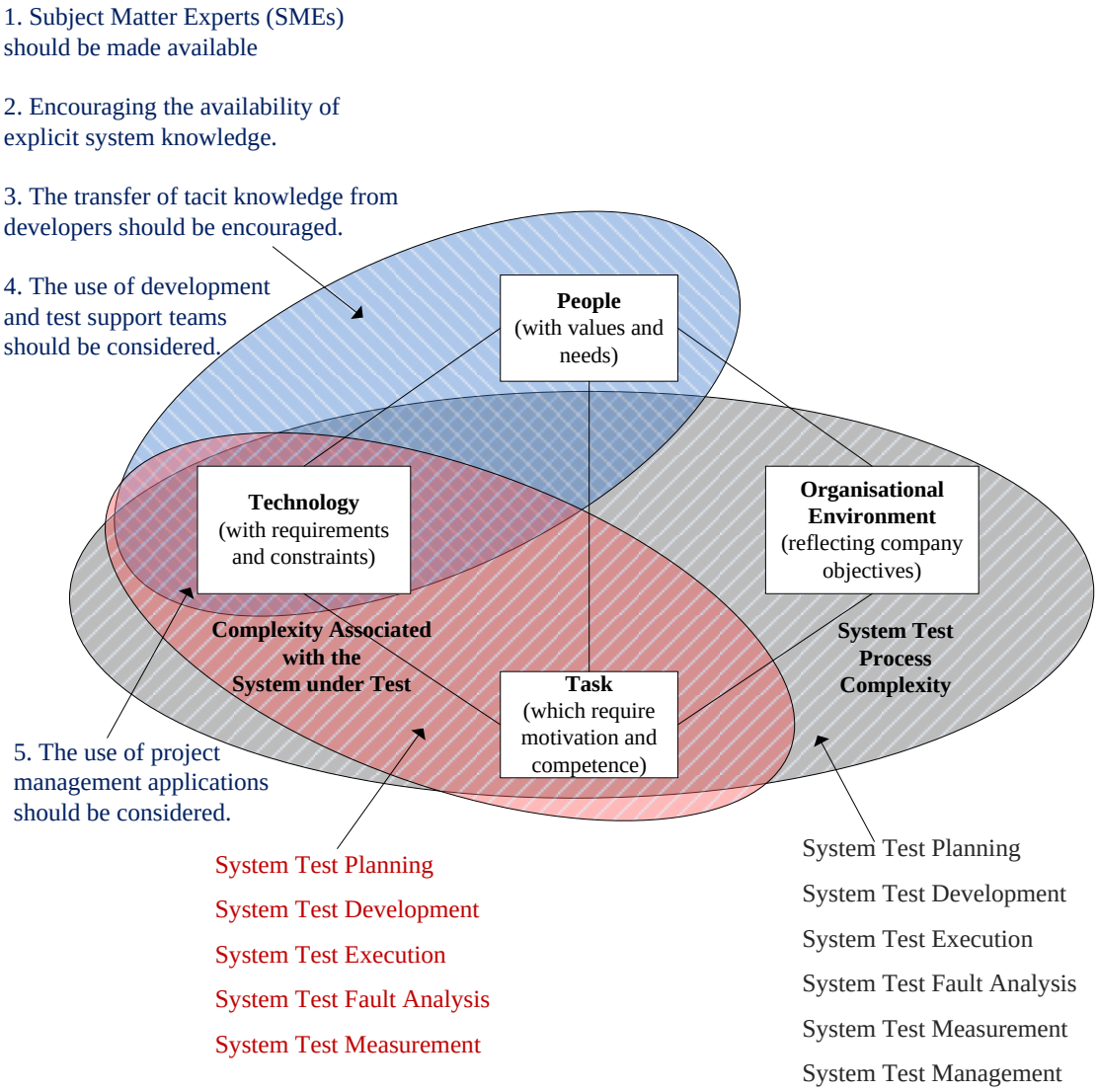


Figure 5.10: A Model of System Test Complexity and Recommended Actions from a Socio-Technical Perspective.

Detailed in figure 5.11, are the stages of system testing (both from a system under test, and from a wider system test perspective) which have shown to have a positive

relationship between complexity and tacit knowledge. Stages relating to the system under test have been detailed in red, whereas stages associated with the wider system test process have been detailed in grey. Also detailed are actions which can be taken to reduce the effects of such complexity (numbered and detailed in blue). It was found that both the system under test, and the wider system test process, were affected by complexity from a technological e.g. inherent complexity related to the system under test (associated with system interoperability and interdependencies), and task perspective e.g. manual testing or manual test measurement. The wider system test process appeared to be additionally impacted from an organisational environment perspective e.g. balancing resources and time to market pressures.

Regarding the suggested actions which could be taken in an effort to reduce the effects of system test complexity, these appeared to primarily relate to people e.g. subject matter experts, and technological e.g. project management applications. As part of further efforts to understand the actions which have been detailed, there is a benefit in applying the views of Hedesstrom (2000). This allows us to further categorise the underlying knowledge, enabling us to differentiate between:

- *Tacit knowledge which has not been formalised because of cost or time limitations.*
- *Tacit knowledge which has not been formalised because of the form of the knowledge, such as embodied knowledge.*

There would appear to be at least some knowledge which falls into the category of knowledge which could be made explicit due to time or cost limitations i.e. *explicit system knowledge* e.g. specifications etc. and knowledge made explicit through the *use of support applications.* Contrary to this, knowledge has also been identified relating *subject matter experts (SMEs)* and *development team members*, of which some at least, falls into the category of knowledge which has not been formalised because of the form of such knowledge.

The analysis which has been presented as part of this chapter will be applied to the research hypotheses in the following concluding chapter.

6 Conclusion

A primary objective of this research is to replicate or extend emergent theory relating to the effect of complexity on the software development process, specifically focussing on the system testing phase. Andrade et al. (2013) have referred to the increasing complexity associated with software testing related tasks, an important aspect of software verification and validation. This research is focussed on the software testing of complete software systems, or *system testing*, as performed by independent test teams. This is distinct from a more granular approach to software testing, which may be carried out as part of *module* or *unit testing*. The use of independent test teams have been endorsed by Talby et al. (2006), who have stated that independent testers allow a more comprehensive test coverage, especially in the case of complex development projects. The primary activities associated with software testing, have been identified by Eickelmann & Richardson (1996), and Desai and Shah (2011). These relate to:

1. *Test Planning* includes the development of a plan relating to test case development. This plan provides an outline of *test objectives*. Detailed as part of test planning are features of the system to be tested, risk assessment issues, organizational training needs, required and available resources, a comprehensive test strategy, resource and staffing requirements, roles and responsibilities, and the overall schedule. Development of a *test architecture*, which involves the identification of required and available resources, is also carried out at this stage.
2. *Test Development* is essentially the development of a *test approach* which includes the specification and implementation of a test configuration. The output of this stage are the test suites, including individual test cases, test input criteria, test documentation, and test adequacy criteria.
3. *Test Execution* includes the execution of the instrumented source code and recording of execution traces. The output of this stage includes test output results, test execution traces, and test statuses.
4. *Test Failure Analysis* includes behavior verification, and the documentation of test execution pass/fail statistics and test failure reports.
5. *Test Measurement* includes test coverage measurement and analysis. Source code is described a typical instrument used to collect execution traces.

Executed test runs have associated with them test coverage measures and test failure measures.

6. *Test Management* includes support for the complete test infrastructure, along with the state preservation of the test environment. Test process automation usually requires a repository of the test infrastructure.

Another important aspect of this research is the relationship between tacit knowledge and system test related complexity. Whereas *explicit knowledge* is stated as having universal character, employed consciously, and not tied to any particular context, *tacit knowledge* is described as being tied to actions, procedures, commitments, ideals, values and emotions, with a strong relationship to past experiences, true beliefs, and the actions of intuition, and implicit rules of thumb (Nonaka & Von Krogh, 2009). Cataldo and Ehrlich (2012) have referred to the lack of existing research which examines both the communication structures facilitating the transfer of knowledge (something which is considered key in software development processes), and also the overall achievement of software development goals, such as productivity or quality. A case for further research into the topic of knowledge, including tacit knowledge, and software engineering, has been made by Ryan and O'Connor (2009), Von Krogh (2012), and Dingsøyr and Šmite (2014). The subject of knowledge as it may apply to the task of system testing, has been discussed by Desai and Shah (2011), and Mantyla and Lassenius (2012).

Taking the aforementioned views into account (and the view of others detailed in chapter three), the following two primary considerations were identified for this research:

1. *Complexity associated with the task of system testing.*
2. *The relationship between system test complexity and tacit knowledge.*

The first consideration of this research i.e. *complexity associated with the task of system testing*, was analysed further in keeping with the views of McKeen et al. (1994), and Brooks(1995), with a further distinction being made between *system complexity* and *task complexity*:

- *Complexity associated with the system under test.*
- *Complexity associated with the process of software development*.

The concept of tacit knowledge, an important aspect of the second research consideration detailed above, along with system test complexity, has been discussed in detail as part of chapter three. Important in this case are the views of Hedesstrom (2000), whose work helps to reconcile the work of Polanyi (1966), Nonaka and Von Krogh (2009), and Tsoukas (2002). He states that the views relating to the aforementioned authors can be encapsulated, by distinguishing between:

- *Tacit knowledge which has not been formalised because of cost or time limitations.*
- *Tacit knowledge which has not been formalised because of the form of the knowledge, such as embodied knowledge.*

Hedesstrom (2000) has made reference to the acceptance amongst a growing number of authors, regarding the clear distinction between tacit knowledge and explicit knowledge. This was important consideration in the development of hypotheses for this research. The following section presents the proposed hypotheses and the research findings. This is followed by a research conclusion, with the final sections of this chapter dealing with research limitations and future research considerations.

6.1 Summary of Findings

As a result of the discussions which were carried out in chapter two and chapter three, the following hypotheses were put forward in chapter four for further investigation:

1. The process of system testing (comprising of *test case planning*, *test case development*, *test case execution*, *test case fault analysis*, *test case measurement*, and *test case management*), is directly affected by complexity associated with the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit

knowledge. It is also proposed that most of this tacit knowledge does not lend itself to being made explicit.

2. The process of system testing (comprising of test case planning, *test case development*, *test case execution*, *test case fault analysis*, *test case measurement*, and *test case management*), is affected by other sources of complexity, independent of the system under test. There exists a positive relationship, with an increase in complexity leading to an increase in tacit knowledge. It is proposed that most of this tacit knowledge does lend itself to being made explicit.

The following sections provide an overview of analysis which has been conducted as part of chapter five and chapter six, relating to these hypotheses.

6.1.1 Observations Relating to the First Hypothesis

After analysing the data acquired through field research (detailed in the preceding chapters), evidence of a positive relationship between complexity associated with the system under test, and tacit knowledge, was shown to exist. Evidence of this relationship is detailed in figure 6.1, through the identification of system test related activities which are affected by complexity, and which displayed a corresponding relationship to tacit knowledge.

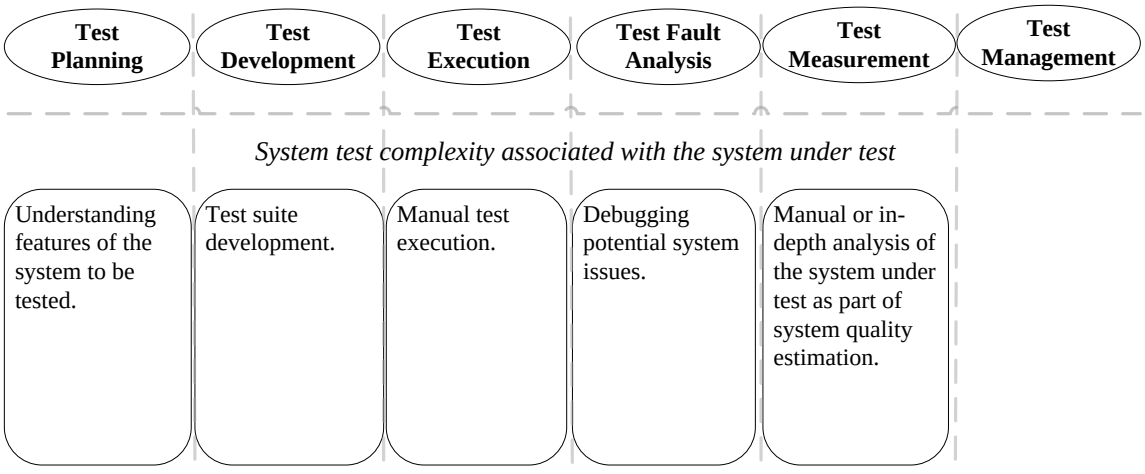


Figure 6.1: Complexity Associated with First Hypothesis.

From the analysis conducted, all stages of system testing, with the exception of *test management*, displayed a positive relationship to tacit knowledge. Complexity associated with the system under test, impacts *system test planning*, by affecting one's ability to understand all aspects of the system to be tested, and an appreciation for how it should be tested. The view was commonly expressed that complexity at this planning stage can have a knock on effect on the subsequent stages of system testing. The implementation of test cases, carried out as part of *test case/suite development*, is also impacted, as distinct from the *development of the test environment*, which is another important aspect of the test development stage. Complexity may come as part of a required understanding of system interactions, something which may be necessary as a result of a manual approach to *test execution* being taken (such as may be taken as part of load or stress testing). The *fault analysis* stage has also been found to be affected by complexity associated with the system under test. This impacts one's ability to effectively carry out root cause analysis of issues, and something which in turn brings a dependency on both test team members and development team members, regarding expertise and knowledge associated with the system under test in practice. The *test measurement* stage was also found to be potentially complex, depending on the level of analysis which is conducted as part of an estimation of system quality. Understandably, complexity is reduced significantly if a more straight forward test measurement approach is adopted, such as the assessment of system quality based on a collection of simple pass or fails, directly relating to test case execution success or failure.

Detailed in figure 6.2 are actions related to the system under test, which have some relationship to explicit knowledge.

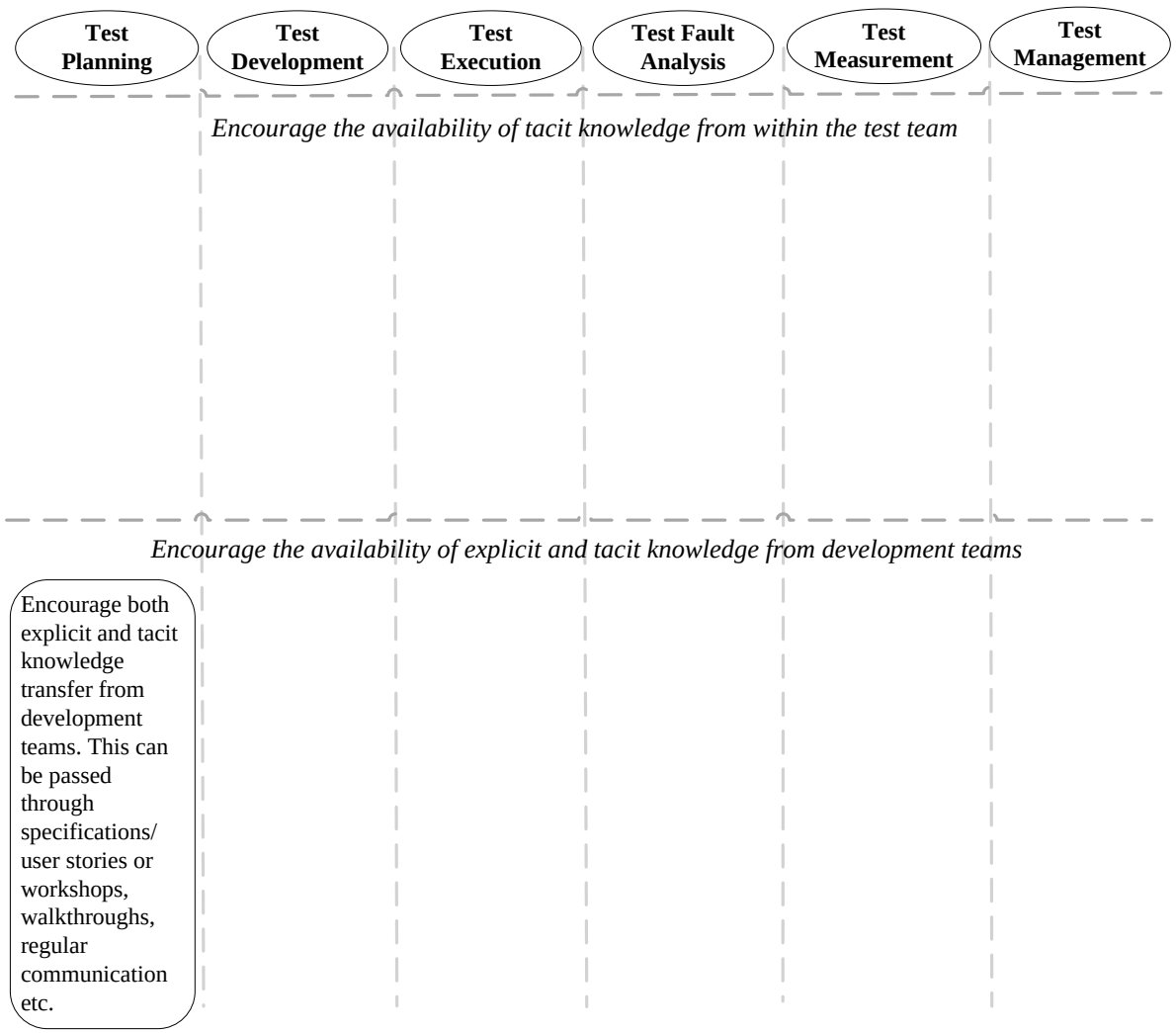


Figure 6.2: Explicit Knowledge Actions Relating To the First Hypothesis.

Numerous authors have referred to the benefits associated with attempting to make knowledge within an organisation explicit and available ((Basili, Lindvall, & Costa, 2001), (Hansen, Nohria, & Tierney, 1999), (Ryan & O'Connor, 2009), (Dingsøyr & Šmite, 2014)). It appears that there is certain knowledge relating to the system under test which can indeed be formalised as explicit knowledge. Such knowledge can be made explicit in the form of specifications or user stories, which are usually created by development teams. The concept of knowledge which may be formalised as explicit knowledge is something which has been put forward by Hedesstrom (2000), in line with the views of Nonaka and Von Krogh (2009), and Polanyi (1966), and is a concept which is applied as part of research by Murphy (2014). This benefit of making available, explicit knowledge relating to the system under test has been emphasised by numerous research participants. However, the sentiment was also expressed that the

benefit of system related specifications in reducing complexity associated with system testing, is diminished if the functional specifications are incomplete, subject to change, or arrive late in the software development process. It was found that the level of documentation associated with a development project does help reduce complexity, but enterprise systems are described as often being very complex by their very nature, with only a certain amount of such knowledge lending itself to being made explicit and documented. The aforementioned findings are considered as supporting this particular hypothesis.

Figure 6.3 highlights additional actions which can be taken as part of efforts to reduce the effects of complexity associated with the system under test, through enabling the flow of tacit knowledge.

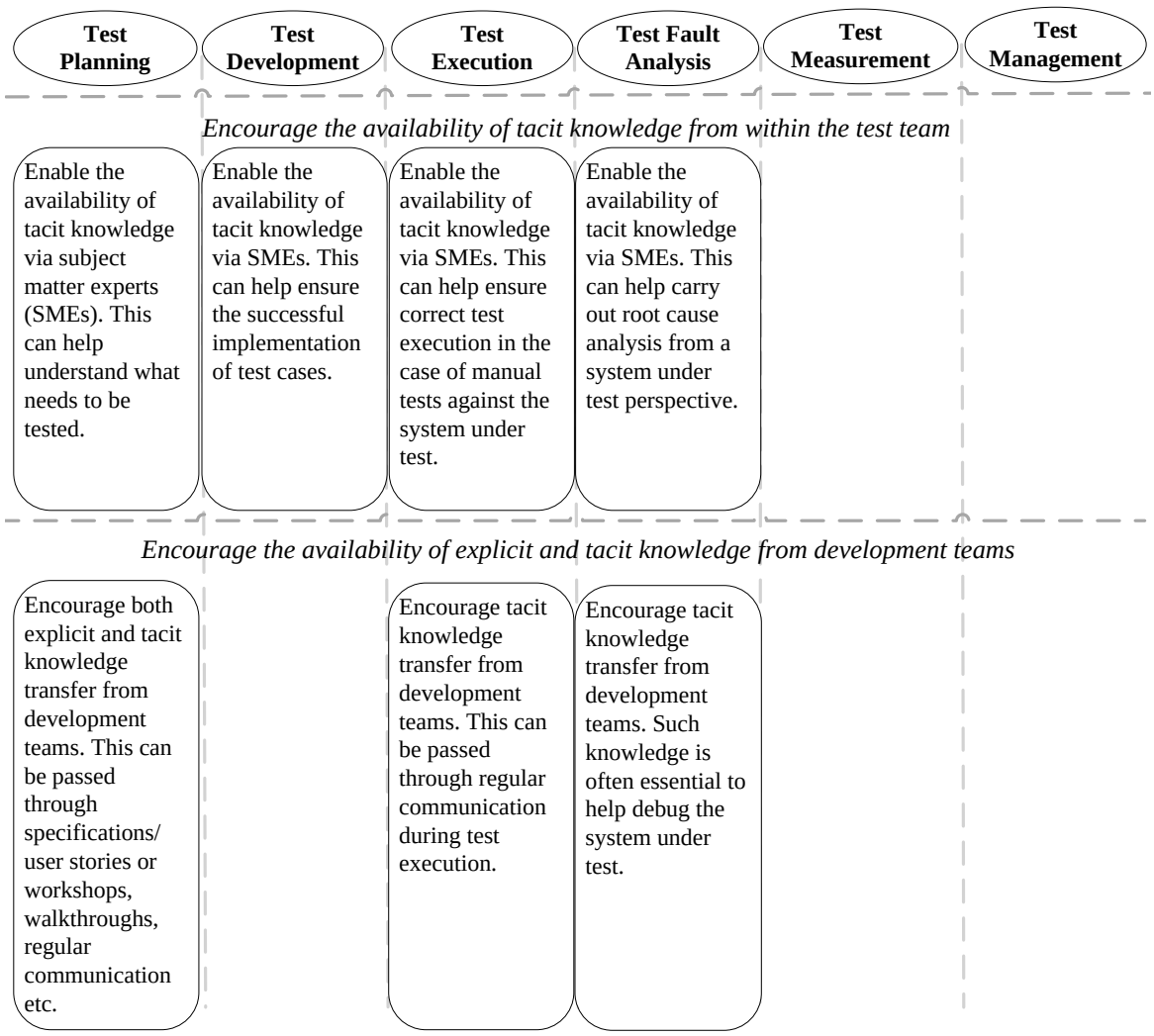


Figure 6.3: Tacit Knowledge Actions Relating To the First Hypothesis.

Support demanded of test and development teams, primarily relates to *test planning*, *test execution*, and *test fault analysis* stage, with the *test development* stage primarily demanding the support of the test team in order to assist the implementation of test cases. This would appear reasonable, given that development team may have an involvement at the subsequent test execution stage, and thus can provide feedback and test validation, if necessary, at that particular stage of the process. The support of development teams has been emphasised by numerous participants, with the common view being expressed that the level of documentation does indeed help reduce complexity. However, enterprise systems are described as often being very complex, with only a certain amount of such knowledge lending itself to being made explicit and documented. This is very much in line with the views of Heddestrom (2000) regarding tacit knowledge which does not lend itself to being easily formalised, due to the form of such knowledge.

A significant amount of time is spent trying to acquire tacit knowledge from development teams, especially in relation to system interaction and expected outcomes under different conditions. If the knowledge is not freely flowing, then this can make the process a lot more inefficient and complex. At the *test planning* stage this knowledge can be transferred via workshops, walkthroughs, and regular communication etc. Regular communication can assist tacit knowledge transfer at the *test execution* and *test fault analysis* stages also. The importance of the distribution of knowledge amongst team members, particularly in the case of complex tasks, has previously been highlighted (Staats, Valentine, & Edmondson, 2010).

Figure 6.4 provides a high level view of the complexity and actions which have been proposed relating to first hypothesis, from a socio-technical perspective.

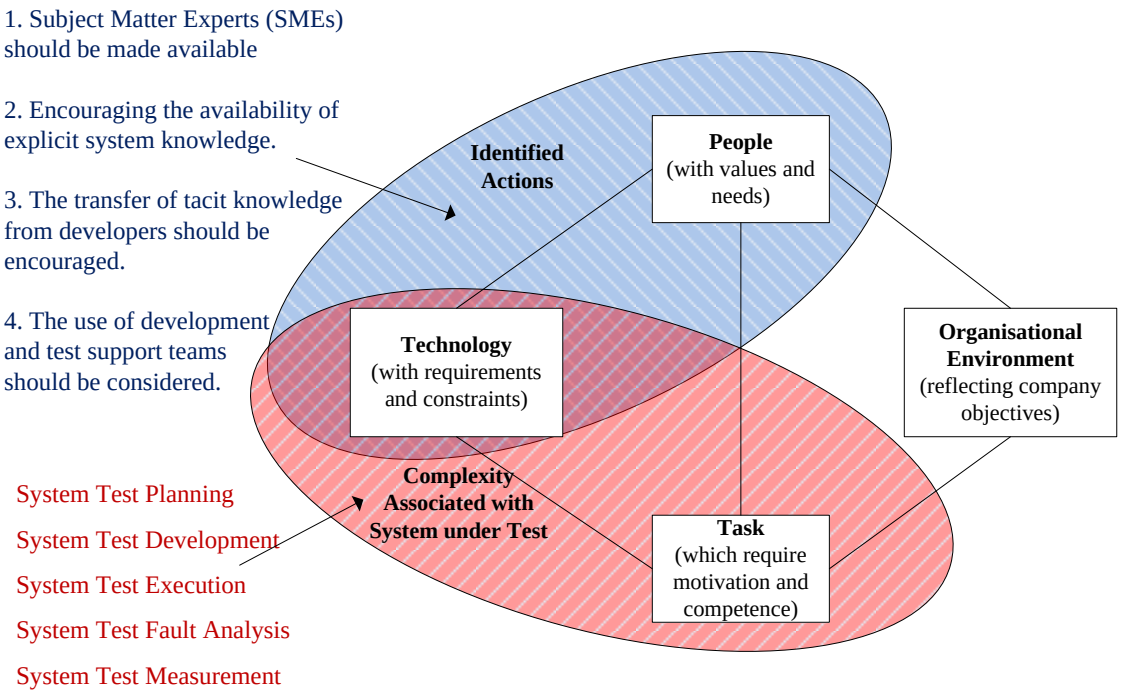


Figure 6.4: The First Hypothesis from a Socio-Technical Perspective.

Included in figure 6.4 are the stages of system testing which displayed evidence of being affected by complexity (detailed in red), and actions which have been proposed to reduce the effects of such complexity (numbered and detailed in blue). The system test activities which have been identified relate to *task* and *technology*, with no obvious link to *people* or *organisational environment*. The actions are associated with enabling the availability of either explicit or tacit knowledge. Sources of tacit knowledge have been identified *as test team members*, *development team members*, and *subject matter experts* (SMEs), with *development team members* having also been identified as an important source of explicit knowledge. The identified actions relate to *people* interaction in the case of test or development team members, or *technology* in the case of explicit knowledge relating to specifications etc.

The following section provides conclusions linked to the second hypothesis.

6.1.2 Observations Relating to the Second Hypothesis

The second hypothesis, relates to the identification of complexity associated with the wider process of system testing, and not directly associated with the system under test in practice. Evidence of such complexity which was found as part of analysis conducted in the preceding chapters, is highlighted in figure 6.5.

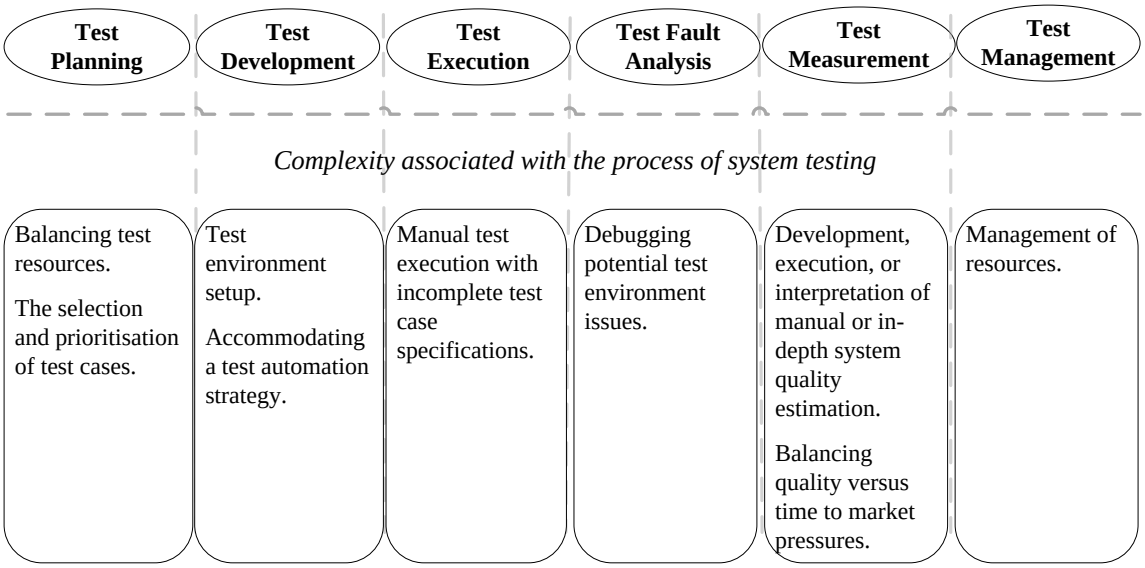


Figure 6.5: Sources of Complexity Associated with the System under Test.

A lack of system understanding can influence one's ability to carry out an estimation of necessary resources to meet test requirements, a necessary aspect of *test planning*. The view was also expressed that complexity at the planning stage can affect one's ability to develop a test strategy i.e. the specification of appropriate tests, and the appropriate selection and prioritisation of test cases (the criteria for the selection of test cases has been referred to as including coverage criteria, resource constraints, and fault detection capability (Lin, Chou, Lai, Huang, & Chung, 2012)). There is evidence that *Test development* is affected by complexity associated with the implementation of a test environment. This stage may also be impacted by the accomodation of an automation strategy, which may not always be a good fit, given time, cost, or quality considerations. Implementation of automation may take a longer initial setup time than manual testing, and may not necessarily work as originally planned. *Test execution* can be complex, if being approached from a manual perspective, and not with the benefit

of an automated test environment. *Test fault analysis* is affected when attempting to eliminate the involvement of the test environment, as part of root case analysis, after the test execution stage has completed. Complexity can also come with the estimation of system quality against expected quality, carried out as part of the *test measurement* stage. The last stage, *test management*, which includes the balancing available resources associated with the required test environment, and enabling test environment preservation, can also prove to be a complex stage.

Figure 6.6 details actions associated with the wider process of system testing which are associated with explicit knowledge.

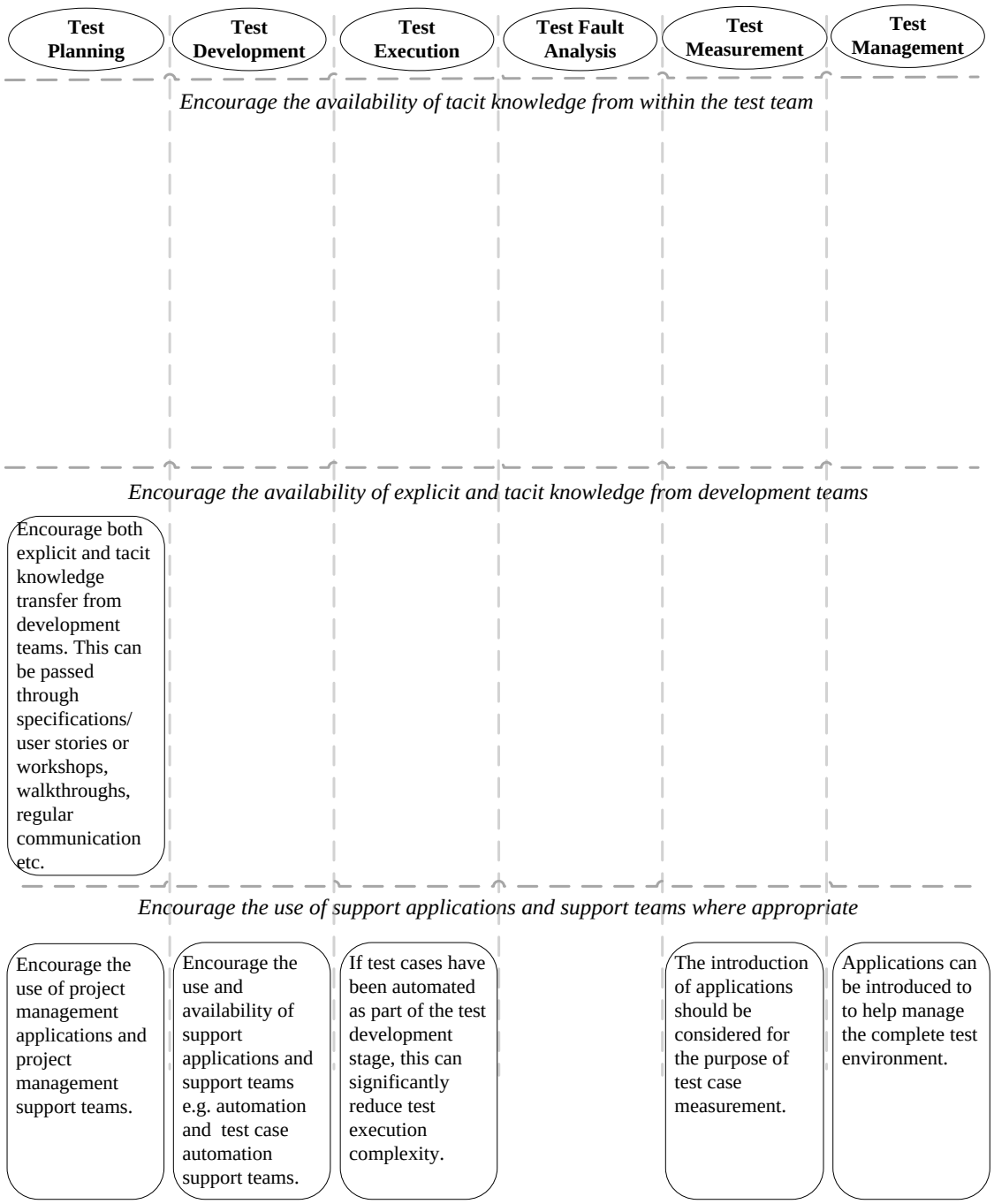


Figure 6.6: Explicit Knowledge Actions Relating To the Second Hypothesis.

Development teams, test support teams, and support applications, all play an important role in the flow and management of explicit knowledge. Important support regarding the system under test would appear to come from development teams, in the form of system related specifications, functional specifications, design specifications, user stories etc. and system deployment knowledge. Such knowledge is essential to enable

effective planning of necessary test resources, and in enabling effective test prioritistion and the selection of appropriate test cases. Project management support teams can also aid at the planning stage through facilitating the acquisition of system and final deployed environment knowledge, thus helping to bridge that knowledge gap between testers and developers. Knowledge can be made explicit via support applications such as test case automation. This can assist the *text execution* stage significantly. It must be noted however, that the use of support teams, such as automation teams, at the *fault analysis* stage of testing, can actually introduce complexity, making it sometimes difficult to quickly determine whether an issue relates to the system under test, or the actual test environment. Applications can also be of benefit at the *test measurement* and *test management* stages, providing automated test measurement, and automated test environment management.

Figure 6.7 highlights actions which can be taken to facilitate the transfer of tacit knowledge associated with the wider system test process.

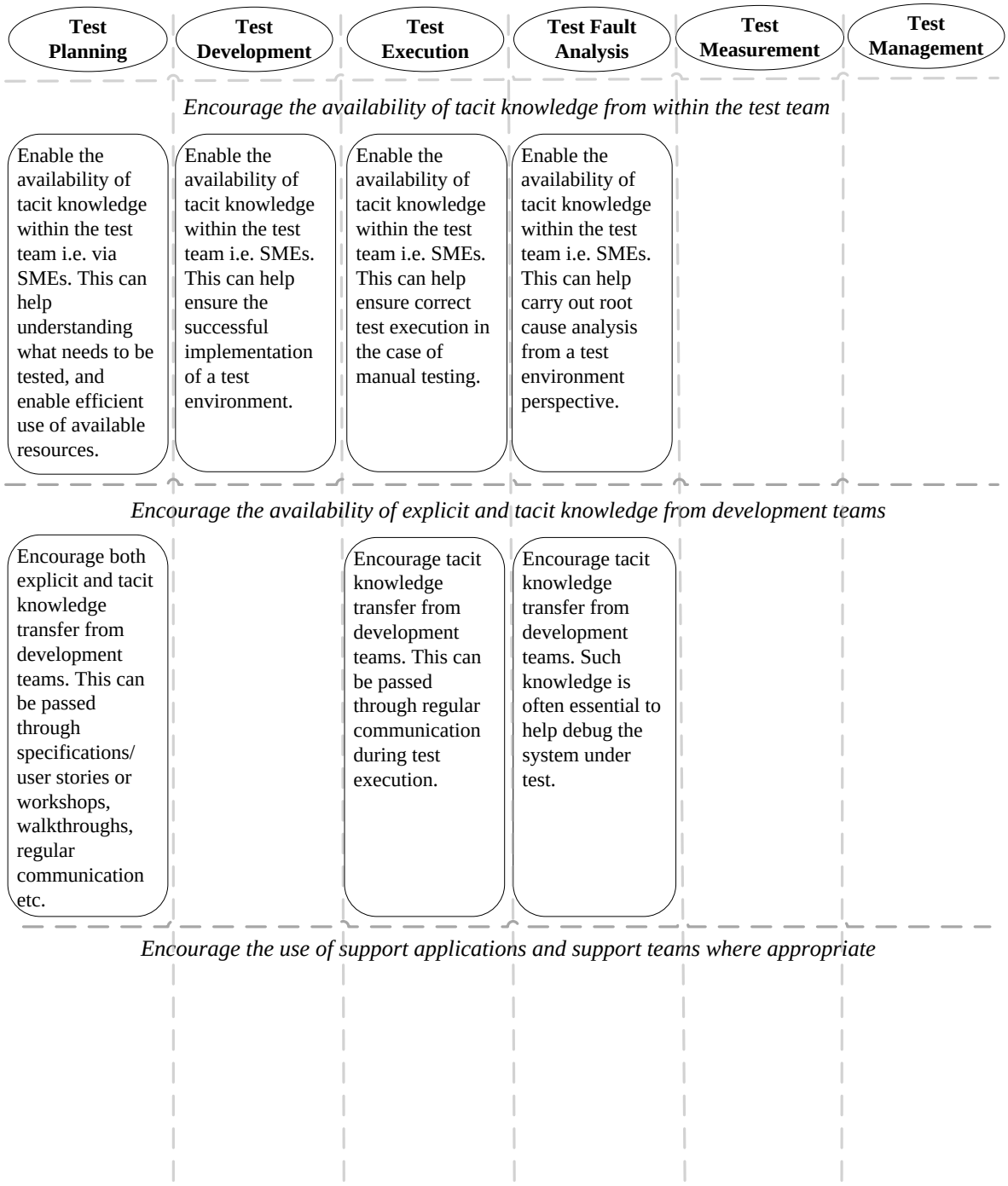


Figure 6.7: Tacit Knowledge Actions Relating To the Second Hypothesis.

Tacit knowledge transfer was described by numerous research participants as being essential in reducing system test complexity. This transfer can be between subject matter experts available to the test team (SMEs) or development team members. Knowledge from within the test team can help achieve a balance with resources at the

test planning stage, and also provide knowledge relating to exactly what can and should be tested, enabling the effective prioritisation and selection of test cases. This knowledge can also help the implementation of a test environment, as part of *test development*, including helping to clarify what can and should be automated. Test team support can assist manual *test execution*, whereby it is necessary to have a detailed knowledge of tests which are being executed, and the correct procedure for execution. At the *fault analysis* stage, one needs to have available requisite knowledge, to be in a position to rule out the involvement of the test environment, after a test case failure.

Support from development teams can help the reduction of complexity at the *test planning*, *test execution*, and *test fault analysis* stages. As part of the planning stage, tacit knowledge can be passed via workshops, walkthroughs, and regular communication etc. At the test execution stage, development support can help ensure that tests are being executed by the test environment correctly. Another important aspect to development support is that it also provides essential expertise at the *fault analysis* stage, helping to debug and validate the performance of the test environment, after test execution.

Figure 6.8 provides a high level view of the complexity and actions relating to second hypothesis, which have been proposed from a socio-technical perspective.

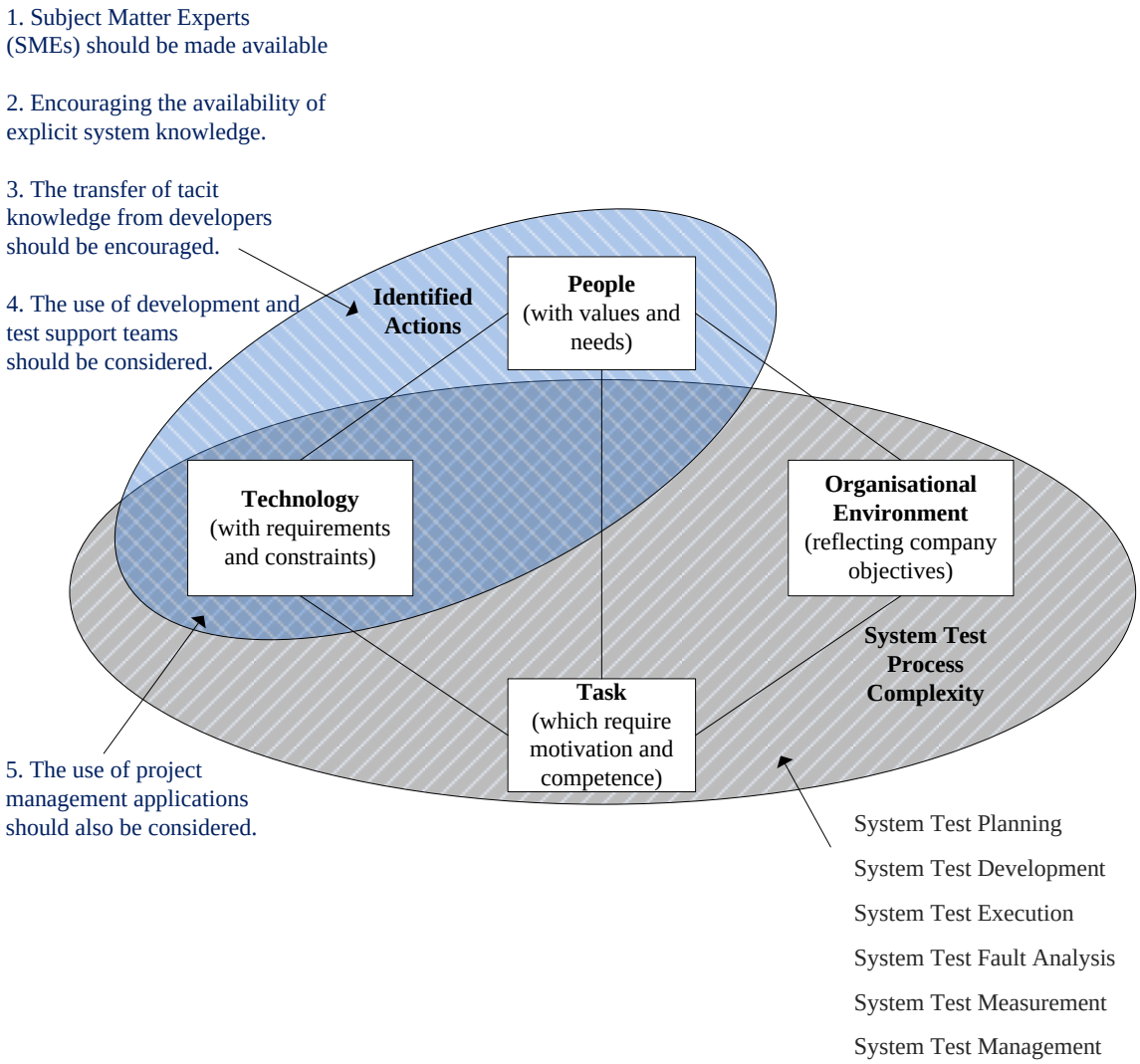


Figure 6.8: The Second Hypothesis from a Socio-Technical Perspective.

It has been found that there is considerable benefit from enabling the availability of tacit knowledge via appropriate *people*, which have been detailed in figure 6.8. Such people can be SMEs or development team members, which have been made available to test team members. Interestingly in this case, is the extent to which explicit knowledge can also play in reducing the effects of complexity. A certain amount of actions which have been detailed, have a link from a *technology* perspective e.g. management applications.

Key to this hypothesis, is that a certain amount of knowledge relating to the process of system testing, does appear to lend itself to being made explicit, whether through the

use of applications, such as project management, an automated test setup, test measurement applications, or through system related specifications. Also interesting are the concerns which have been highlighted relating to system test automation, which can be complex to implement effectively and efficiently, but can lead to significant benefit at the test execution stage, if implemented effectively.

The following section offers a conclusion for this research.

6.2 Concluding Discussion

This section provides an overview of the research which has been conducted, while also detailing considerations for software development practices. Figure 6.9 details the test activities which have been identified in previous sections as being affected by complexity, and which have a direct relationship to tacit knowledge. The model details activities form both a *system under test* (system in practice), and from a *wider system test process* perspective.

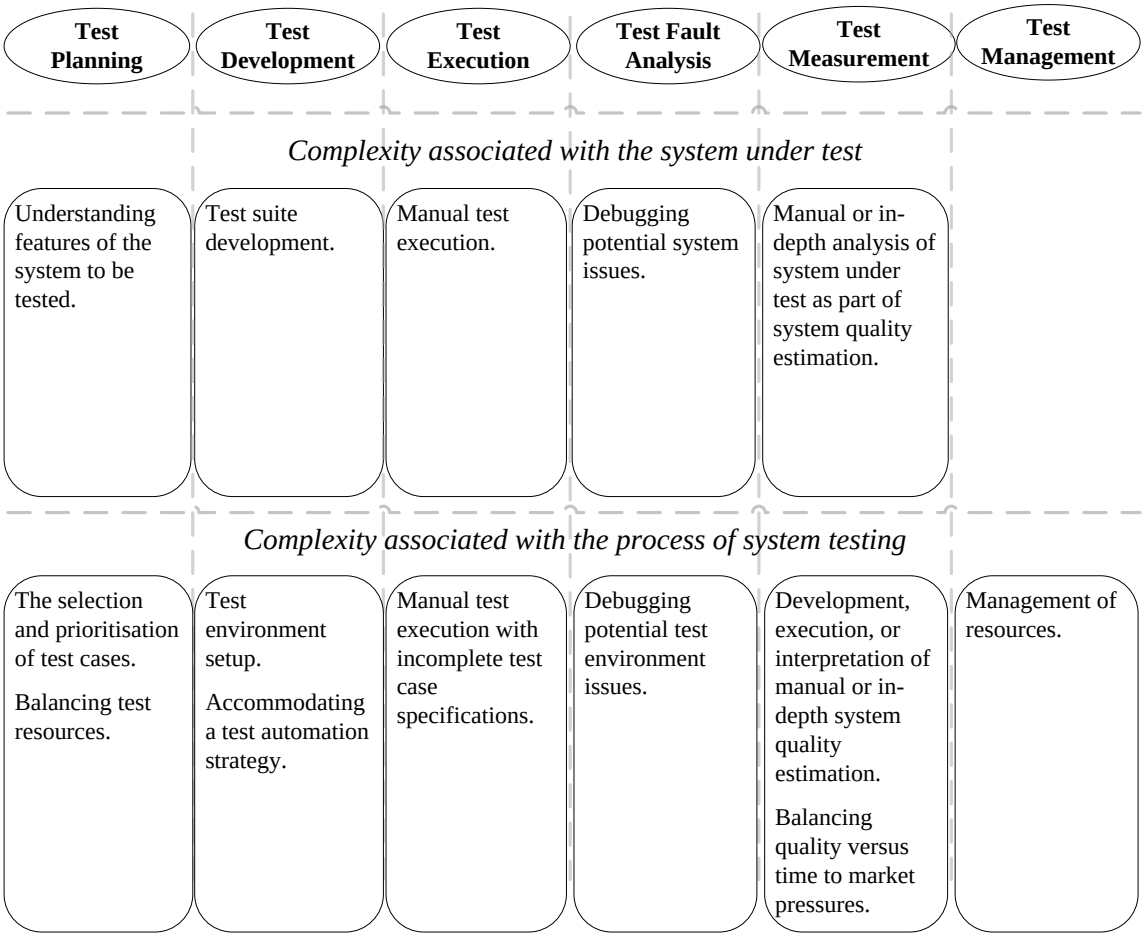


Figure 6.9: Concluding Model of System Test Complexity with a Relationship to Tacit Knowledge.

Table 6.1 provides a comparison between the actions detailed in figure 6.9, against the software testing functions as outlined by Eickelmann & Richardson (1996), and Desai and Shah (2011).

Comparison to Complete Set of Test Functions (Eickelmann & Richardson (1996))

Test Planning	Test Development	Test Execution	Test Fault Analysis	Test Measurement	Test Management
Balancing functional v non-functional objectives.	Implementation of a test approach i.e. a complete test configuration (facilitating white box or black box).	Test execution against system under test.	Test result verification.	Test coverage measurement.	Consideration of the test architecture and test environment preservation.
Understanding features to be tested.	Development of test suites.	Test artefact recording i.e. test output results, test traces, test status.	Test result analysis and documentation (pass/fail, test coverage).	Test failure measurement.	Maintenance of test resource repository (necessary in the case of an automated test process).
Achievemet of risk assessment.					
Facilitating training requirements.					
Balancing necessary vs available resources (both human and technical).					
Development of test strategy (test selection, minimisation and prioritisation).					
Allocation of roles and responsibility.					
Schedule development.					

Table 6.1: Comparison to Complete Set of Test Functions.

Those activities (or functions) which were identified as part of this research, have been highlighted in red font. A noticeable activity which was not referenced as part of the actions detailed in table 6.1, but which does feature in figure 6.9, is the the reference to *balancing quality versus time to market pressure*, which was categorised as being associated with the test measurement stage. The remaining activities (in black font)

detailed in table 6.1, were not found to have any specific relationship to system test complexity.

As identified in previous sections, actions which should be considered to reduce the effects of complexity associated with the system under test, relate to the transfer of both explicit knowledge and tacit knowledge. Further reference to both system test activities affected by complexity, and actions, from a socio-technical perspective, are detailed in figure 6.10. The effected stages of system testing have been detailed in red (relating to the system under test) and grey (relating to the wider process of system testing). Recommended actions have been numbered and are detailed in blue.

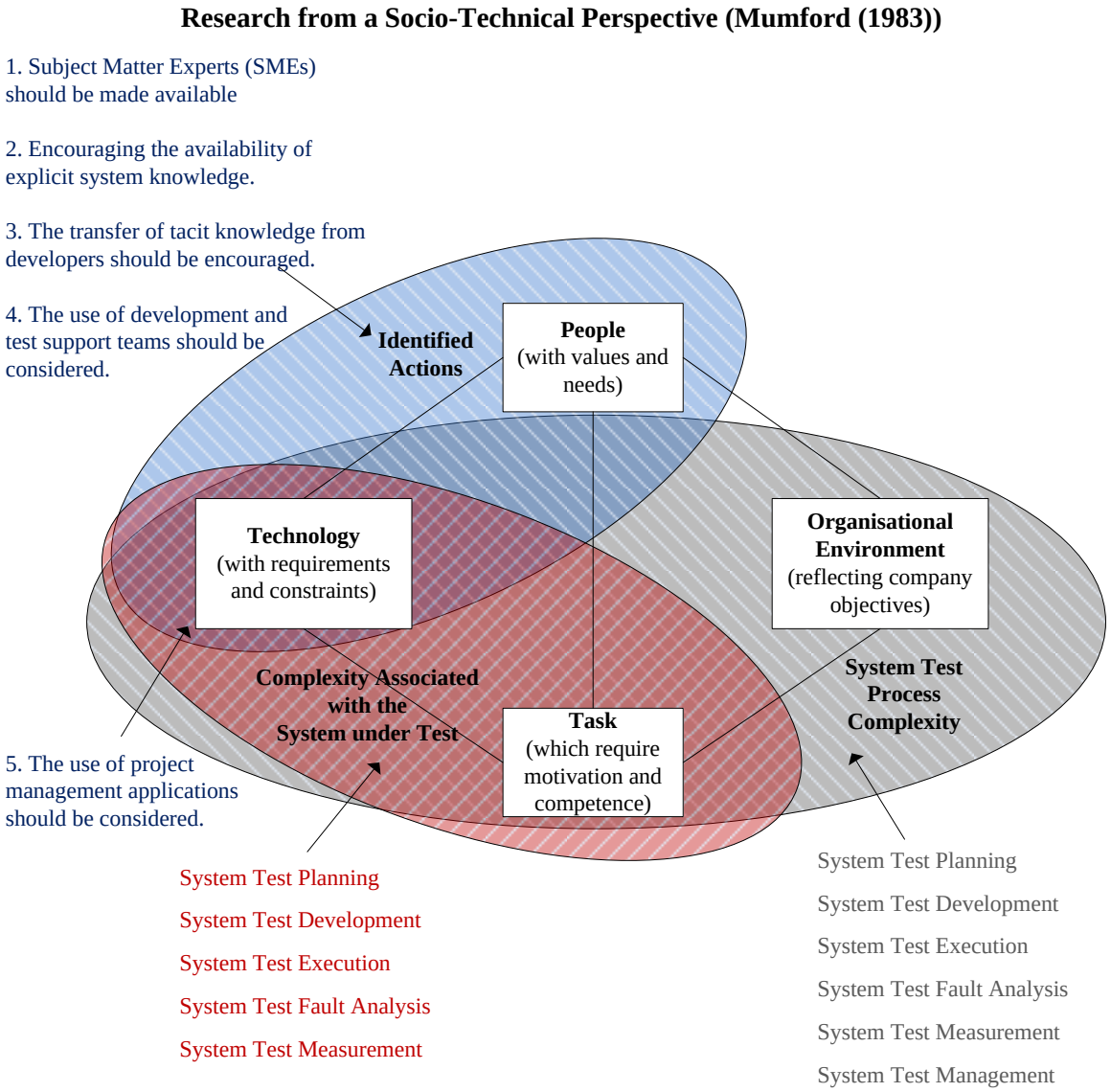


Figure 6.10: Research from a Socio-Technical Perspective.

The model detailed in figure 6.10 is discussed in the following section as part of research implications.

Implications for the Development Process

This research has identified the importance of the availability of both explicit knowledge and tacit knowledge, relating to both the system under test, and associated with the wider process of system testing. A certain amount of knowledge relating to the process of system testing, lends itself to being made explicit, whether through the use of applications, such as project management, automation, or test measurement applications, or through system related specifications, user stories etc. The benefit of enabling the availability of tacit knowledge, via appropriate *people*, has been evident in the case of both complexity related to the system under test, and in the case of complexity associated with the process of system testing. Such people may be *test team accessible SMEs*, or *development team members*. The availability of both explicit knowledge and tacit knowledge has obvious benefit in terms of system quality, through ensuring necessary required knowledge is readily accessible throughout the test process. Such knowledge can also influence the time to market for the system under test, if a lack of access to such knowledge is an impediment to progress of system testing. A lack of access to knowledge could occur as a result of a delay relating to the receipt of developments specifications, a delay in the development of the test environment, or a delay in waiting for a system to be debugged, which may be necessary as part of the fault analysis stage of testing.

Knowledge transfer is an important aspect of software development environments ((Chau & Maurer, 2004), (Joia & Lemos, 2010), (Nidhraa, Yanamadalaa, Afzalb, & Torkara, 2013)). Previous chapters have covered the views of authors such as Chau et al. (2003), Turk et al. (2005), and Moe et al. (2012), who have acknowledged the relationship between the applied development methodology, the approach to knowledge management, and knowledge sharing. Some software development methodologies such as agile, have been described as being heavily reliant on the communication of tacit knowledge via interpersonal contact. Turk (2005) and Dingsøyr and Šmite (2014) have argued that there is an increased importance of tacit

communication via personal contact, given the movement away from traditional development strategies, which have been perceived as rigid, plan driven models (Chau, Maurer, & Melnik, 2003). The success of agile development methodologies is based on team members understanding, experience, and their ability and willingness to share applicable tacit knowledge. This is carried out on a continuous, informal basis, between software development team members, and customers (Turk, France, & Rumpe, 2005).

Turk et al. state that when the team's tacit knowledge is insufficient for the application's life-cycle needs, things work fine, but that there is also the risk that the team will become overly dependent on experts, and may suffer from "corporate memory loss", either of which could result in unrecognized shortfalls in available tacit knowledge. Chau at al. (2003) have referred to traditional software development as striving to achieve idealistic goals via Tayloristic processes. Such traditional models are described as relying on explicit documentation in order to provide the process and product information, to enable team members to effectively achieve their goals (Turk, France, & Rumpe, 2005). Such explicit knowledge reduces the risk of knowledge loss ((Rajagopalan, 2014), (Dingsøyr & Šmite, 2014)).

While the importance of explicit knowledge has been reinforced by this research, there has been a lack of evidence to suggest that the availability of tacit knowledge to test teams is of any less importance to the process of system testing, when operating in a traditional software development environment. The sentiment was commonly expressed by participants, that even though a considerable amount of explicit knowledge relating to the system is freely available, that a good deal of knowledge relating to the system under test, which is demanded for effective system testing, is actually tacit in nature (approximately 60% of participants operating in a traditional development environment, and 60% of participants operating in an agile development environment, expressed similar sentiments). The concept of complexity which is inherent in the system, is a concept which has been referred to by numerous authors ((Mumford, 1983), (Brooks F. , 1995), (Lehman, 1996), (Lyytinen, Mathiassen, & Ropponen, 1998), (Espinosa, Slaughter, Kraut, & Herbsleb, 2007), (de Silva & Balasubramaniam, 2012)). To cater for the availability of tacit knowledge relating to the system under test, and indeed both explicit and tacit knowledge required by system

testing in general, an appropriate knowledge management structure needs to be in place. This would appear to be required, irrespective of the employed development methodology. Research implications, from a knowledge management perspective, are discussed in the following section.

Knowledge Management Considerations

The importance of a knowledge management approach has been emphasised in the previous section. This is supported by Desai and Shah (2011) who state that regardless of the approach to software development, there is necessity to manage knowledge associated with the various stages of software testing i.e. *test planning*, *test development*, *test management*, *test execution*, *test fault analysis* and *test measurement*. Leidner et al. (2008) have stated that organisations traditionally adopt one of two approaches to knowledge management. The first approach involves a focus within the organisation on *communities of practice*, or alternatively, the second approach focuses on facilitating *the process of creation, sharing, and the distribution of knowledge*.

While organisations may adopt different aspects of both approaches, both approaches are claimed to present different challenges. The first approach is said to be cognisant of the fact that a great deal of organisational knowledge is in fact held *tacitly*. Formal processes and technologies are stated as not being suitable for enabling the transmission of such knowledge. The approaches to knowledge management from both a community perspective, and a process perspective, have also been referred to as *personalisation* or *codification* approaches, respectively (Hansen, Nohria, & Tierney, 1999). Facilitating the transfer of tacit knowledge, is of particular importance in the case of a *personalisation*/*communities of practice* approach to knowledge management ((Hansen, Nohria, & Tierney, 1999), (Leidner, Alavi, & Kayworth, 2008)).

The following section provides some common approaches to supporting knowledge management (Dingsøyr & Šmite, 2014), and specifically how different aspects of knowledge management are dealt with in practice (Dorairaj, Noble, & malik, 2012). In a review of empirical studies relating to knowledge management of global software

development projects, Dingsøyr & Šmite have identified the following five common approaches to knowledge management. These approaches provide varying support for personalisation and codification approaches to knowledge management:

1. *Systems school*: this relates to the application of technology for knowledge management e.g. knowledge repositories.
2. *Cartographic school*: this relates to the knowledge maps and the creation of knowledge directories. Such an approach is useful for storing knowledge relating to resources, skills, projects opportunities etc.
3. *Engineering school*: this supports knowledge management through a focus on processes and knowledge flow with organisations. This has been referred to as primarily relating to processes for mapping knowledge, conducting project retrospectives, accomodating mentoring programs, and catering for detail relating to work processes e.g. CMM (the capability maturity model). This model is stated as being primarily based on explicit knowledge.
4. *Organisation school*: this approach is concerned with networks for sharing or pooling knowledge. This is often put into practice by way of *communities of practice* relating to a common topic of interest. It is stated that such communities facilitate the transfer of both tacit knowledge and explicit knowledge. This is typically a less formal approach than in the case of knowledge repositories.
5. *Spatial school*: this approach is related to how an office space can facilitate the knowledge management. This can range from setting up whiteboards, to the use of an open plan office structures to encourage engagement. A popular use in the case of an agile approach to software development, is the use of taskboads, which relate to project status and are visible to stakeholders. This approach is stated as being dependent on the colocation of stakeholders, and appears to work well for smaller teams.

Global organisations employing a more traditional approach to software development are stated as predominantly relying on *systems* or *engineering* schools, whereas those working in accordance with agile methodologies, are stated as relying on *spatial* and *organisational* schools. The *cartographic* school is stated as providing a cost-effective

means of knowledge management, irrespective of the employed development methodology.

Some indication of the relationship between such schools, and knowledge related activities in practice, is provided by Dorairaj et al. (2012). In their analysis of knowledge management approaches, involving 28 agile centred software development companies, the aforementioned authors have highlighted examples of the principal knowledge based activities. These activities have been categorised based on their contribution to *knowledge generation*, *knowledge codification*, *knowledge transfer*, and *knowledge application*. *Knowledge generation*, as described, has at least some relationship to previously mentioned engineering, organisational and spatial schools, as mentioned by Dingsøyr & Šmite (2014). *Knowledge codification* has a relationship to the systems school. *Knowledge transfer*, would appear to have at least some relationship to all of the schools mentioned, similar to *knowledge application,* which is also arguably facilitated by each of the schools, via different approaches.

The following examples have been provided by Dorairaj et al. (2012), regarding these knowledge activities in practice:

Knowledge generation, is stated as being facilitated by:

1. *Project inception*: workshops etc. facilitating the crystalization of ideas between stakeholders and developers.
2. *Customer collaboration*: sources of knowledge relating to the actual required product, in terms of requirements etc.
3. *Formal training*: formal training is stated as enabling the standardisation of training content and practices across multiple sites in an organisation.
4. *Communities of practice*: these consist of self organising groups of individuals who share information, insight, experience, and technical skills on a specialised discipline, and collaborate on common challenges or the stimulation of new ideas.
5. *Self learning*: the encouragement of individuals to learn appropriate to their role, is seen as an important aspect of knowledge generation.

Knowledge codification is stated as being facilitated by mediums such as:

1. *Wikis*: accessible knowledge via wiki pages is seen as an effective method to encourage knowledge sharing and collaboration.
2. *Documentation*: the availability of explicit documentation is seen as crucial to complex software systems which are subject to frequent modification, providing details relating to requirements, specifications, limitations and implementation.
3. *Technical presentation*s: notwithstanding the difficulties associated with sharing ideas, concepts, and technical expertise, through short presentations, there is a distinct benefit in capturing such knowledge for future access.

Knowledge transfer is facilitated by:

1. *Regular development meetings:* meetings such as scrums, where ongoing work is shared and impediments discussed, are seen as beneficial as a team building exercise.
2. *Project inception meeetings*: such meetings involving project managers, technical leads, and business analysts, are seen as beneficial in determining the viability of potential projects. There is further benefit to knowledge, acquired as a result of such exercises, being passed to wider groups on completion.
3. *Pair programming*: the integrative collaboration of developers on projects through pair programming, is stated as having the benefit of increasing knowledge transfer, enhancing learning, and encouraging knowledge creation.
4. *Knowledge management tools*: tools are stated as being readily available off the shelf, and development processes are stated as benefiting from the integration of such tools into development processes, thereby facilitating the capture of knowledge from a variety of sources throughout a project lifecycle.
5. *Face-to-face meetings*: though knowledge transfer can be facilitated through audio or video conferencing, face-to-face meetings are said to have an advantage, especially when dealing with high levels of complexity and ambiguity in a project.
6. *Rotation*: in keeping with the previous comment regarding the benefits of face-to-face meetings, the rotation of team members between different project sites, has been stated as having a benefit in facilitating higher levels of knowledge transfer.

7. *On-site customer visits*: on-site customer visits are stated as driving software development, by continually providing correct and complete understanding of customer needs and requirements, thus adding to knowledge relating to the system deployment and use.
8. *Cross-functional teams*: there is a benefit in grouping teams of developers, analysts, testers, and individuals with other necessary domain expertise who can contribute to the success of a project through communication and collaboration.
9. *Discussion*: discussion with subject matter experts, regardless of geographical location, facilitates openness and communication, and offers further opportunities to generate, refine, and reprioritise, both requirements and specifications.

Key points in terms of *knowledge application* are:

1. *Repository interaction (referred to as "similar context")*: interaction with knowledge management applications such as Wikis, facilitate the flow of knowledge to and from individuals, and the collaborative knowledge stored in the Wiki pages.
2. *Information understanding (referred to as "problem solving")*: although technology can assist with the storage and transfer of knowledge, the knowledge itself can only be created and utilised by individuals, therefore team members need to understand information contained in Wikis etc. in order to create new knowledge, which can in turn help realise solutions to future problems.
3. *Future sprints/projects*: the availability of knowledge from multiple technologies and functional documents, is essential for the completion of complex projects.

As this research supports the necessity for organisations involved in the software development of large enterprise systems, for adopting a combination of both a personalisation approach in the case of tacit knowledge, and codification approach in the case of explicit knowledge, the detailed knowledge management approaches and activities are all of potential benefit. Some stages such as test case planning have been shown to benefit significantly from explicit knowledge, which can be made available

through knowledge codification activities. Such explicit knowledge can take the form of system specifications, user stories etc. Knowledge made explicit in the form of knowledge management tools such as test automation and project management, have also been shown to be beneficial. On the contrary, stages such as the fault analysis stage, would appear to have a stronger link to tacit knowledge, therefore knowledge transfer is a key aspect of this stage. This could be facilitated through knowledge transfer activities relating to the use of cross-functional teams, involving both developers and testers, and through the availability of subject matter experts.

The following section discusses both the limitations and future considerations of this research.

6.3 Limitations and Future Considerations

It must be acknowledged that some concerns have been raised regarding collection models which are employed as part of this research. A fixed-point, survey questionnaire type approach, has been adopted by Pee et al. (2010), Hsu et al. (2011) and Akman et al. (2011), to seemingly good effect. However, such an approach is referred to as lacking in realism of context, and is deemed to be low in precision of measurement (McGrath, 1984). Similar concerns have been raised by Woodside (2009), who state arguments against both questionnaire type approaches, and case study approaches, when adopted in isolation. As an alternative to an independent questionnaire or case study approach, a more open interview approach was taken as part of this research. The critical incident technique (used to good effect by Kaplan and Duchon (1988)), has been employed for this research, facilitating the retrieval of both qualitative data (via a series of open questions) and quantitative data (via Likert scale ratings). A similar unstructured interview approach has previously been used to good effect by Ryan and O'Connor (2009). The approach which has been taken is an attempt to take a balanced approach to evidence gathering, as advocated by both McGrath (1984), and Woodside (2009). This balanced approach was an attempt to mitigate the limitations of the individual collection models.

There may also be a perceived limitation associated with the work environment of the participants involved in this research. The selection of participants was influenced by a desire to include some degree of environmental variation. It has been stated that variation over the population selection can provide control over environmental variation, as well as enabling the definition of limits for the analysis of findings (Eisenhardt, 1989). While environmental variation has been welcomed, it must be acknowledged, that the organisations involved operate in completely different industries, and the participants test completely different software systems, and operate in different work environments. However, the participants are engaged in the testing of enterprise software systems, and it was found that there were relatively high levels of perceived complexity relating to the system under test, across the four organisations, as detailed in Figure 6.11 (these details have been taken from figure 4.5).

Industry	values	Complexity
Enterprise Storage	**Average ratings**	5.8
Test consultancy	**Average ratings**	6.0
Life Assurance	**Average ratings**	6.5
Payroll	**Average ratings**	6.0
	Standard deviation	0.30

Figure 6.11: Complexity Ratings Associated with the System under Test

Although the average ratings of the perceived complexity associated with the system under test are relatively high, and the standard deviation has been deemed acceptable, the fact that there were market and work environment differences between the organisations involved, and these differences have not been considered in terms of this research, could be perceived as a potential research limitation.

In addition to the aforementioned limitations, the following future considerations are also apparent:

1. *The true value of an automation strategy*: Interesting concerns were raised relating to system test automation. This research found that an automated test environment can be complex to implement effectively and efficiently, and to debug, but can lead to significant benefit at the test execution stage. Martin et al. (2007) has carried out some work in this particular area, and has stated that non-functional tests are often tests which do not easily conform to automation. Future research could be carried out regarding the role of automated test environments. One suggested topic could be a cost-benefit analysis associated with pursuing an automation strategy involving global software development projects.

2. *Participant experience*: notwithstanding the fact that the experience of participants has been taken as a limitation, it can also be taken as an opportunity for future research. Andrade et al. (2013) state that experience is an important characteristic of software testing, and there is a benefit relating to experience which has been gained through past projects. Whereas explicit knowledge is stated as having universal character, employed consciously, and not tied to any particular context, tacit knowledge is described as being tied to actions, procedures, commitments, ideals, values and emotions, with a strong relationship to past experiences, true beliefs, and the actions of intuition, and implicit rules of thumb (Nonaka & Von Krogh, 2009). Some quantitative analysis has been conducted from an experience perspective (participants with less than 10 years' experience, and participants with greater than 10 years' experience). While there was some interesting data, relating to some stages of system testing, most notably the test planning, test fault analysis, *test measurement* and *test management stages*, which displayed a stronger relationship between system test complexity and tacit knowledge, with relation to inexperience testers, there was also notable discrepancies with this analysis in comparison to the qualitative data (similar to those highlighted in section 5.3.2). Thus, there is an opportunity for further research to be carried out in this area.

3. *The effects of task familiarity on system testing*: Banker and Slaughter (2000) have stated that that task familiarity is increasingly important in larger software tasks, and Espinosa et al. (2007) have stated that as task familiarity increases, software development time decreases, proportionally. Task familiarity, as it may apply to the task of software system testing, has not been taken account of as part of this research. This also leaves an opportunity for future research to be conducted in this area, as it might apply to system test complexity.

7 Bibliography

1. Açıkgöz, A., Günsel, A., Bayyurt, N., & Kuzey, C. (2013). Team Climate, Team Cognition, Team Intuition,and Software Quality: The Moderating Role of Project Complexity. *Science+Business Media*, pp.(1145–1176) (Vol.23).
2. Agresti, W. (2003). Tailoring IT Support to Communities of Practice. *IT-Pro*, 24-28.
3. Ahmad, M., Mawarny, R., Abdulah, M., Omar, M., Ahmad, K., & Abbas, M. (2012). Measuring tacit knowledge acquired during problem based learning teaching method in learning management system environment. *AWERProcedia Information Technology & Computer Science* (pp. pp.(775-781)). AWERProcedia Information Technology & Computer Science.
4. Akman, I., Misra, S., & Cafer, F. (2011). The Role of Leadership Cognitive Complexity in Software Development Projects: An Empirical Assessment for SImple Thinking. *Human Factors and Ergonomics in Manufacturing & Service Industries*, pp.(516-525) (Vol.21; No.5).
5. Alur, R., & Dill, D. (1994). A Theory of timed automa. *Theoretical Computer Science*, pp.181-235 (Vol.126).
6. Andrade, J., Ares, J., Martínez, M., Pazos, J., Rodríguez, S., Romera, J., et al. (2013). An architectural model for software testing lesson learned systems. *Information and Software Technology*, pp 18-34 (Vol.55).
7. Ashby, W. (1956). An Introduction to Cybernetics. In W. Ashby, *The Law of Requisite Variety* (pp. pp.(206-218)). London: Chapman and Hall Ltd.
8. Baccarini, D. (1996). The concept of project complexity - A review. *International Journal of Project Management*, pp.201-204.
9. Baig, M., & Khan, A. (2010). A Formal Technique for Reducing Software Testing Time Complexity. *Innovations and Advances in Computer Sciences and Engineering*, pp.(197-201).
10. Banker, R., & Slaughter, S. (2000). The moderating effects of structure on volatility and complexity in software enhancement. *Information Systems Research*, pp.219-240 (Vol.11;No.3).
11. Basili, V., & Perricone, B. (1984). Software errors and Complexity: An Empirical Investigation. *Communications of the ACM*, pp.42-52 (Vol.27;No.1).

12. Basili, V., Lindvall, M., & Costa, P. (2001). Implementing the Experience Factory concepts as a set of Experience Bases. *13th International Conference on Software Engineering and Knowlwedge Engineering* (pp. pp.(102-109)). Knowledge Systems Institute.
13. Baskerville, R. (2006). Artful Planning. *European Journal of Information Systems*, pp.113-115.
14. Beath, C. (1987). Managing the User Relationship inInformation Systems Development Projects: A Transaction Governance Approach. *Proc. 8th International Conf. on information Systems*, (pp. 415-427). Pittsburgh.
15. Beck, K. (1999). Embracing change with extreme programming. *IEEE Computer*, Chaptars 6,17.
16. Beck, K. (2000). Extreme Programming Explained. Boston: Addison-Wesley.
17. Bentley, L., & Whitten, J. (2007). *Systems Analysis & Design for the Global Enterprise.* McGraw-Hill.
18. Berlo, D. (1960). *The Process of Communication.* New York: Holt, Rinehart, and Winston, Inc.
19. Berman, O., & Cutler, M. (1998). Optimal Software Implementation Considering Reliability and Cost. *Computer Ops Res.*, pp.857-868 (vol. 25; No. 10).
20. Berman, O., & Cutler, M. (2004). Resource Allocation during tests for optimally reliable software. *Computers & Operations Research*, pp.1847-1865 (Vol 31).
21. Beynon, W., Boyatt, R., & Chan, Z. (2008). Intuition in Software Development Revisited. *20th Annual Psychology of Programming Interest Group Conference*. Lancaster: University of Warwick.
22. Beynon-Davis, P. (1995). Information Systems Failure; The Case of LAS-CAD Project. *European Journal of Information Systems*, pp.171-184.
23. Bhadoriya, N., Mishra, N., & Malviya, A. (2014). Agile Software Development Methods, Comparison with Traditional Methods & Implementation in Software Firm. *International Journal of Engineering Research & Technology*, pp. (1656-1662) (Vol.3; No.7).
24. Bhattacharya, P., Iliofotou, M., Neamtiu, I., & Faloutsos, M. (2012). Graph-Based Analysis and Prediction for Software Evolution. *Proceedings of the 2012 International Conference on Software Engineering* (pp. pp.(419-429)). Institute of Electrical and Electronic Engineers.

25. Boehm, B. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer 21*, pp.61-72.
26. Boehm, B. (2002). Getting ready for agile methods, with care. *IEEE*, pp.64-69.
27. Boehm, B., & Turner, R. (2004). Balancing agility and discipline: Evaluating and Integrating Agile and Plan Driven Methods. *Proceedings of the 26th International Conference on Software Engineering (ICSE'04).* Institute of Electrical and Electronics Engineers.
28. Brooks, F. (1995). *The Mythical Man-Month.* Addison-Wesley.
29. Brooks, F. P. (1986). No Silver Bullet: Essence and Accidents of Software Engineering. *Proceedings of the IFIP Tenth World Computing Conference*, (pp. 1069-1076).
30. Brown, S., Venkatesh, V., & Goyal, S. (2011). Expectation Confirmation in Technology Use. *Information Systems Research*, pp.(1-14) .
31. Burnstein, I., Suswanassart, T., & Carlson, C. (1996). The Development of a Testing Maturity Model. *Proceedings of the Ninth International Quality Week Conference.* San Francisco.
32. Cai, K. (1998). On Estimating the Number of Defects Remaining in Software. *Journal of Systems and Software*, pp.93-114 (Vol.40;No.2).
33. Cai, K., & Card, D. (2008). An analysis of research topics in software engineering - 2006. *The Journal of Systems and Software*, pp.1051-1058 (Vol 81).
34. Campbell, D. (1988). Task Complexity: a review and analysis. *Academy of Management Review*, pp. 40-52 (Vol.13).
35. Casti, J., & Karlqvist, A. (1986). *Complexity, Language, and Life: Mathematical Approaches.* Berlin;Heidelberg;NewYork;Tokyo: Springer-Verlag.
36. Cataldo, M., & Ehrlich, K. (2012). The Impact of the Structure of Communication Patterns on New Product Development Outcomes. *ACM*, pp.3081-3090.
37. Catelani, M., Ciani, L., Scarano, V. L., & Bacioccola, A. (2010). Software automated testing: A solution to maximize the test plan coverage and to. *Computer standards and interfaces*, pp.152-158 (Vol.33).
38. Cenfetelli, R., & Bassellier, G. (2009). nterpretation of Formative Measurement in Information Systems Research. *MIS Quarterly*, pp. 689-707 (Vol. 33; No. 4).
39. Charmaz, K. (1995). Grounded Theory. In J. Smith, R. Harré, & L. Lanenhove, *Rethinking Methods In Psychology* (pp. 27-49). London: Sage Publications Ltd.

40. Chau, T., & Maurer, F. (2004). Knowledge sharing in Agile Software Teams. *Proceedings of the Symposium on Logic Versus Approximation* (pp. 173–183). Berlin Heidelberg: Springer Verlag.
41. Chau, T., Maurer, F., & Melnik, G. (2003). Knowledge Sharing: Agile Methods vs. Tayloristic Methods. *Enabling Technologies: Infrastructure for collaborative Enterprises* (pp. 302-307). IEEE.
42. Chen, T., & Lau, M. (1998). A new heuristic for test suite reduction. *Information and Software Technology*, pp.347-354.
43. Chen, T., & Lau, M. (1998b). A simulation study on some heuristics for test suite reduction. *Information and Software Technology*, pp.777-787.
44. Chen, Y., Rosenblum, D., & Vo, K. (1994). A system for selective regression testing. *Proceedings of the 16th International Conference on Software Engineering* (pp. 211-220). Institute of Electrical ad Electronics Engineers.
45. Cho, V., & Wright, R. (2010). Exploring the evaluation framework of strategic information systems using repertory grid technique: a cognitive perspective from chief information officers. *Behaviour and Information Technology*, pp.447–457 (Vol. 29, No. 5).
46. Chou, S., & He, M. (2004). Knowledge management: the distinctive roles of knowledge assets in knowledge creation. *The Journal of Information Science*, pp146-164, (Vol.30; No.2).
47. Clarke, P., & O'Connor, R. (2012). The situational factors that affect the software development process: Towards a comprehensive reference framework. *Journal of Information Software and Technology*, pp 433-447 ((Vol. 54; No. 5).
48. Cockburn, A. (2001). Agile Software Development. Reading, MA: Addison-Wesley.
49. Collins, H., & Kusch, M. (1998). The shape of actions: What humans and machines can do. *The MIT Press*.
50. Connelly, C., Zweig, D., Webster, J., & Trougakos, J. (2012). Knowledge Hiding in Organizations. *Journal of Organizational Behavior*, pp.(64–88) Vol. 33.
51. Cooprider, J., & Henderson, J. (1991). Process Fit: Perspectives on Achieving Prototyping Effectiveness. *Journal of Management Information Systems*, pp.67-87.
52. Crispin, L., & Gregory, J. (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams.* Crawfordsville, Indiana: Addison-Wesley.

53. Cule, P., Schmidt, R., Lyytinen, K., & Keil, M. (2000). Strategies for heading off IS project failure. *Information Systems Management*, pp.65-73.
54. Curtis, B., Kellner, M., & Over, J. (1992). Process Modelling. *Communications of the ASM*, pp.75-90.
55. Curtis, B., Krasner, H., & Iscoe, N. (1988). A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*, pp.1268-1287.
56. Daft, R., Lengel, R., & Trevino, L. (1987). 'Message equivocality, media selection and manager performance: implications for information systems. *MIS Quarterly*, pp.355-366 (Vol.11; No. 3).
57. Dane, E., & Pratt, M. (2007). Exploring Its Role in Management Decision Making: The Role of Intuition. *Academy of Management Review*, pp. 44-54 (Vol.32; No.1).
58. Davis, M., Challenger, R., Jayewardene, D., & Clegg, C. (2013). Advancing socio-technical systems thinking: A call for bravery. *Applied Ergonomics*, pp.(1-10).
59. day, R. (2005). Clearing up "implicit knowledge": Implications for knowledge management, information science, psychology, and social epistemology. *Journal for American Society for Information Science and Technology*, pp(630-635).
60. de Silva, L., & Balasubramaniam, D. (2012). Controlling software architecture erosion: A Survey. *The Journal of Systems and Software*, pp.132-151.
61. Debbarma, M. K., Singh, N. P., Shrivastava, A. K., & Mishra, R. (2011). Analysis of Software Complexity Measures for Regression Testing. *Proceedings of International Conference on Advances in Computer Engineering*, (pp. 88-92).
62. Delahaye, M., Kosmatov, N., & Signoles, J. (2013). Common Specification Language for Static and Dynamic Analysis of C Programs. *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (pp. pp 1230-1235). New York: ACM.
63. Desai, A., & Shah, S. (2011). Knowledge Management and Software Testing. *International Conference on Emerging Trends in Technology* (pp. 767-770). Mumbai: ACM.
64. Desmoulin, A., & Viho, C. (2007). Automatic Interoperability Test Case Generation based on Formal Definitions. *Proceedings of the 12th International Workshop on Formal Methods for Industrial Critical Systems FMICS'07* (pp. 234-250). Berlin Heidelberg: Springer-Verlag.
65. Deutsch, M. (1949). A Theory of Cooperation and Competition. *Human Relations*, pp.(129-152).

66. Diamantopoulos, A., & Siguaw, J. (2006). Formative Versus Reflective Indicators in Organizational Measure Development: A Comparison and Empirical Illustration. *British Journal of Management*, pp.263-282 (Vol.17).
67. Diamantopoulos, A., Riefler, P., & Roth, K. (2008). Advancing formative Measurement Models. *Journal of Business Research*, pp. 1203-1218 (Vol.61; No.12).
68. Dingsøyr, T., & Šmite, D. (2014). Managing Knowledge in Global Software Development Projects. *IT Professional*, pp. (22-29) (No.01; Vol.16).
69. Dorairaj, S., Noble, J., & malik, P. (2012). Knowledge management in Distributed Agile Software Development. *2012 Agile Conference* (pp. pp.(64-73)). CPS.
70. Dorairaj, S., Noble, J., & Malik, P. (2012). Knowledge Management in Distributed Agile Software Development. (pp. pp 64-73). 2012 Agile Conference.
71. Drucker, P. (1999). Knowledge-worker productivity: The biggest challenge. *California Review Management*, pp.79-94 (Vol 41, no.2).
72. Dyba, T., & Dingsøyr, T. (2008). Empirical studies of Agile Software Development. *Information and Software Technology*, pp.833-859 (Vol.50;No.9/10) .
73. Edwards, J., & Bagozzi, R. (2000). On the Nature and Direction of Relationships Between Constructs and Measures. *Psychological Methods*, pp.155-174 (Vol.5; No.2).
74. Eickelmann, N., & Richardson, D. (1996). An Evaluation of Software Test Environment Architectures. *Proceedings of the 18th International Conference on Software Engineering* (pp. 353-364). Berlin: IEEE.
75. Eisenhardt, K. (1989). Building Theories From case Study Research. *The Academy Of Management Review*, pp. (532-550) (Vol.14; No.4).
76. En-Nouaary, A. e. (1998). Timed Test cases generation based on characterization technique. *In Real-time System Symposium, IEEE Computer Society*, (pp. 220-229). Madrid.
77. Espinosa, J., Slaughter, S., Kraut, R., & Herbsleb, J. (2007). Familiarity, Complexity, and Team Performance in Geographically Distributed Software Development. *Oraganisational Science*, pp. 613–630 (Vol. 18, No. 4).
78. Faraj, S., & Sproull, L. (2000). Coordinating expertise in software development teams. *Management Science*, pp.1554-1568 (Vol.46;No.12) .

79. Farr, W. (1996). Handbook of Software Reliability Engineering. (pp. 71-117). McGraw-Hill.
80. Fenton, N. E., & Ohlsson, N. (2000). Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, pp.797-814 (Vol. 26; No. 8).
81. Ferrer, J., Chicano, F., & Alba, E. (2013). Estimating software testing complexity. *Information and Software Technology*, pp 2125–2139 (Vol.55).
82. Fitzgerald, K., Seale, N., & Kerins, C. (2008). The Critical Indicent Technique: A Useful Tool for Conducting Qualitative Research. *Journal of Dental Education*, pp.(299-304) (Vol.72; No.3).
83. Flanagan, J. (1954). The critical incident technique. *Psychological Bulletin*, pp.327-358 (Vol. 51).
84. Ford, K., Petry, F., Adams-Webber, J., & Chang, P. (1991). An Approach to Knowledge Acquisition Based on the Structure of Personal Construct Systems. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, pp.78-88 (Vol. 3; No. 1).
85. Frederiksen, N., Saunders, D. R., & Wand, B. (1957). The In-Basket Test. *Psychological Monographs: General and Applied*, Vol. 71, No. 9.
86. Glaser, B., Strauss, A., & Anselm, L. (1967). *The Discovery of Grounded Theory:Strategies for Qualitative Research.* Chicago: Aldine.
87. Gottfredson, L. (2002). Dissecting practical intelligence theory its claims and evidence. *Intelligence*, pp.1–55 (Vol. 30).
88. Grambow, G., Oberhauser, R., & Reichert, M. (2015). Context-Aware and Process-Centric Knowledge Provisioning: An Example from the Software Development Domain. *Intelligent Systems Reference Library*, pp 179-209 (Vol.95).
89. Guinan, P., Cooprider, J. G., & Faraj, S. (1998). Enabling software development team performance during requirements definition: A behavioural versus technical approach. *Information Systems Research*, pp.101-125 (Vol 9, No. 2.) .
90. Hair, J., Ringle, C., & Starstedt, M. (2011). PLS-SEM: Indeed a Silver Bullet. *Journal of Marketing Theory and Practice*, pp. 139–151; (Vol. 19; no. 2).
91. Hansen, M., Nohria, N., & Tierney, T. (1999). What's your strategy for managing knowledge? *Harvard Business Review*, pp(1-11); Mar-April.

92. Harold, M., & Soffa, M. (1989). Interprocedural data flow testing. *TAV3 Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis and verification* (pp. 168-167 (Vol.14;No.8)). New York: ACM.
93. Harrold, M., Gupta, R., & Soffa, M. (1993). A methodology for controlling the size of a test suite. *ACM transactions on software engineering and methodology (TOSEM)*, pp.270-285 (Vol.2;No.3).
94. Hedesstrom, T. a. (2000). What is meant by tacit knowledge? Towards a better understanding of the shape of actions. *8th European Conference on Information Systems, 3-5 July 2000, Vienna, Austria*. Vienna, Austria: LSE (London School of Economics and Ploitical Science).
95. Herbsleb, J. (2007). The future of socio-technical coordination. *Future of software engineering (IEEE)*, pp.(188-198).
96. Highsmith, J. (2000). Adaptive Software Development: A Collaborative Approach to Managing Complex Systms. New York.
97. Highsmith, J., & Cockburn, A. (2001). Agile Software Development: The Business of Innovation. *IEEE Computer Society*, pp.120-127 (Vol.34;No.9).
98. Holste, J., & Fields, D. (2010). Trust and tacit knowledge sharig and use. *Journal of knowledge management*, pp.128-140 (Vol.14;No.1).
99. Holzworth, D. P., Huth, N. I., & deVoil, P. (2011). Simple software processes and tests improve the reliability and usefulness of a model. *Environmental modelling & software*, pp.510-516 (Vol.26).
100. Horgan, J., & Mathur, A. (1996). *Handbook of Software Reliability Engineering*. McGraw-Hill.
101. Hsu, J., Chang, J., Klein, G., & Jiang, J. (2011). Exploring the impact of team mental models on information utilization and project performance in system development. *International Journal of Project Management*, pp.(1-12) (Vol.29).
102. Huo, M., Verner, J., Zhiu, L., & Bahar, M. (2004). Software Quality and Agile Methods. *Proceedings ofthe 28th Annual International Computer Sofwtare and Applicatios Conference* (pp. 520-525 (Vol. 1)). The Computer Society.
103. Iacobucci, D. (2010). Structural equations modeling: Fit Indices, sample size, and advanced topics. *Journal of Consumer Psychology*, pp 90-98 (Vol. 20).
104. IEEE. (1990). *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York.

105. Jackson, M. (2006). What Can we Expect From Program Verification? *IEEE Computer Society*, pp. 65-71.

106. Jamwal, D. (2010). Analysis of Software Quality Models for Organizations. *International Journal of Latest Trends in Computing*, pp.19-22 (Vol.1;No.2).

107. Jankowicz, A. (2004). *The easy guide to the repertory grids.* Chichester: Wiley.

108. Jick, T. (1979). Mixing Qualitative and Quantitative Methods: Triangulation in Action. *Administrative Science Quarterly*, pp.602-611 (Vol.24;No.4).

109. Jinzenji, K., Hoshino, T., Williams, L., & Takahashi, K. (2012). Metric-based quality evaluations for iterative software development approaches like Agile. *23rd International Symposium on Software Reliability Engineering Workshops* (pp. pp.(54-63)). IEEE.

110. Johnson, D., & Johnson, R. T. (2005). New Developments in Social Interdependence Theory. *Genetic, Social, and General Psychology Monographs*, pp.(285-358).

111. Joia, L. A., & Lemos, B. (2010). Relevant factors for tacit knowledge transfer within organisations. *Journal of Knowledge management*, pp.410-427 (vol.14;No.3).

112. Jones, C. G., Gray, G. L., Gold, A. H., & Jones, C. (2010). Strategies for improving systems development project success. *Issues in Innformation Systems*, pp.164-173 (vol11;No.1).

113. Kaplan, B., & Duchon, D. (1988). Combining Qualitative and Quantitative Methods in Information Systems Research: A Case Study. *MIS Quarterly*, pp.571-586 (Vol.12;No.4).

114. Keil, M. (1995). Pulling the Plug: Software Project Management and the Problem of Project Escalation. *MIS Quarterly*, pp.421-448 (Vol.19;No.4).

115. Keil, M., Cule, P., Lyytinen, K., & Schmidt, R. (1998). A framework for identifying software project risks. *Communications of the ACM*, pp.76-83 (Vol.41, No.11).

116. Kelly, D. (2008). Innovative Standards For Innovative Software. *Computer*, pp.88-89 (Vol.41; No.7).

117. Kelly, G. (1955/1991). *The Psychology of Personal Constructs, vols. 1 and 2.* Routledge.

118. Khan, M., & Khan, F. (2014). Importance of Software Testing in Software Development Life. *International Journal of Computer Science Issues*, pp 120-123 (Vol.11, Issue 2, No.2).

119. Kim, G., Shin, B., & Grover, V. (2010). Investigating The Contradictory Views of Formative Measurement in Information Systems Research. *MIS Quarterly*, pp. 345-366 (Vol. 34; No. 2).

120. Kimble, C. (2013). Knowledge management, codification and tacit knowledge. *Information Research*, pp (1-14) (Vol.18;No.2).

121. Ko, A. J., DeLine, R., & Venolia, G. (2007). Information Needs in Collocated Software Development Teams. *29th International Conference on Software Engineering (ICSE'07)* (pp. pp. 344-353). Washington D.C.: Institute of Electrical and Electronic Engineers.

122. Ko, A., Abraham, R., Beckwith, L., Blackwell, A., & Burnett, M. (2011). The State of the Art in End-User Software Engineering. *ACM Computing Surveys (CSUR)* , pp 1-61 (Vol.43;No.21).

123. Kochhar, P., Bissyand, T., Lo, D., & Jiang, L. (2013). An Empirical Study of Adoption of Software Testing in Open Source Projects. *Quality Software International COnference* (pp. pp.(103-112)). Najing : IEEE .

124. Kothari, A., Rudman, D., Dobbins, M., Rouse, M., Sibbald, S., & Edwards, N. (2012). The use of tacit and explicit knowledge in publichealth: a qualitative study. *Implementation Science*, pp (1-12) (Vol.7; No.20).

125. Kuhn, D., Wallace, D., & Gallo Jnr., A. (2004). Software Fault Interactions and Implications for Software Testing. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pp 1-4 (Vol.30;No.6).

126. Lam, A. (1997). Embedded firms, embedded knowledge: Problems of collaboration and knowledge transfer in global cooperative ventures. *Organisational Studies*, pp.973-996 (Vol.18; No.6).

127. Larman, C. (2004). *Agile and Iterative Development.* Addison-Wesley.

128. Leavitt, H. (1964). *Applied Organizational Change in Industry: Structural, Technical, and Human Approaches in New Perspectives in Organizational Research.* Chichester,: Wiley.

129. Lee, G., & Xia, W. (2010). Toward Agile: An integrated analysis of quantitative and qualitative field data on sofwtare development agility. *MIS Quarterly*, pp.87-114 (Vol.34;No.1).

130. Lee, G., Delone, W., & Espinosa, J. (2006). Ambidextrous Coping Strategies In Globally Distributed Software Development Projects. *Communications of the ACM*, pp.35-40 (Vol.49,No.10).

131. Lee, G., Espinosa, J., & DeLone, W. (2013). Task Environment Complexity, Global Team Dispersion, Process Capabilities, and Coordination in Software Development. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pp 1751-1771 (Vol.39).

132. Lehman, M. (1996). Laws of Software Evolution Revisited. *Lecture Notes in Computer Science*, pp.108-124 (Vol. 1149/1996).

133. Leidner, D., Alavi, M., & Kayworth, T. (2008). The Role of Culture in Knowledge Management: A Case Study of Two Global Firms. In D. Leidner, M. Alavi, & T. Kayworth, *Global Information Systems: The Implications of Culture for IS Management* (pp. pp.(263-287)). London: Elsevier Butterworth-Heinemann.

134. Leung, H., & White, L. (1991). A cost model to compare regression test strategies. *Software Maintenance* (pp. 201-208). Institute of Electrical and Electronics Engineers.

135. Lewin, K. (1935). *A Dynamic Theory of Personality*. New York: McGraw-Hill.

136. Li, Y., Chen, Y., Liu, J., Cheng, Y., Wang, X., Chen, P., et al. (2011). Measuring task complexity in information search from user's perspective. *Proceedings of the American Society for Information Science and Technology*, pp.(1-8) (Vol.38; No,1).

137. Lin, Y.-D., Chou, C.-H., Lai, Y.-C., Huang, T.-T., & Chung, S. (2012). Test coverage for large code problems. *The Journal of Systems and Software*, pp.16-27.

138. Littlewood, B., Popov, P., & Strigini, L. (2002). Assessing the reliability of diverse fault-tolerant software-based systems. *Safety Science*, pp.781-796 (Vol.40).

139. Liu, J. Y.-C., Chen, V. J., Chan, C.-L., & Lie, T. (2008). The impact of software process standardization on software flexibility and project management performance. *Information and Software Technology*, pp.889-896 (Vol.50;No9/10).

140. Loveland, S., Miller, G., Prewitt, R. J., & Shannon, M. (2005). *Software testing techniques: finding the defects that matter*. Hingham, Massachusetts: Charles River Media, Inc.

141. Lowry, P., & Gaskin, J. (2014). Partial Least Squares (PLS) Structural Equation Modeling. *IEEE TRANSACTIONS ON PROFESSIONAL COMMUNICATION*, pp.123-146 (Vol. 57; No. 2).

142. Lu, Y., Xiang, C., & Wang, X. (2011). What affects information systems development team performance? An exploratory study from the perspective of combined socio-technical theory and coordination theory. *Computers in Human Behavior*, pp.(811-822).

143. Lyu, M. (1996). *Handbook of software reliability engineering.* McGraw-Hill.

144. Lyytinen, K., Mathiassen, L., & Ropponen, J. (1998). Attention Shaping and Software Risk - A Categorical Analysis of Four Classical Risk Management Approaches. *Information Systems Research*, pp.233-255 (Vol.9;No.3).

145. Mantyla, M., & Lassenius, C. (2012). The Role of the Tester's Knowledge in Exploratory Software Testing. *Software Engineering, IEEE Transactions on*, pp (707-724) (Vol.39; No.5).

146. Mantyla, M., Itkonen, J., & Iivonen, J. (2012). Who Tested my Software? Testing as an Organizationally Coss-Cutting Activity. *Software Quality Journal*, pp.(145–172) (vol.20).

147. Marciniak, R., Amrani, R., Rowe, F., & Adam, F. (2014). Does ERP integration foster Cross-Functional Awareness? Challenging conventional wisdom for SMEs and large French Firms. *Business Process Management journal*, pp. 865-886 (Vol.20;No.6).

148. Markus, L., & Keil, M. (1994). If We Build It, They Will Come. *Sloan Management Review*, pp.11-25.

149. Martin, A., Biddle, R., & Noble, J. (2009). The XP Customer Team: A Grounded Theory. *Agile Conference* (pp. 57-64). IEEE Computer Society.

150. Martin, D., Rooksby, J., Rouncefield, M., & Sommerville, I. (2007). 'Good' Organisational Reasons for 'Bad' Software Testing: An Ethnographic Study of Testing in a Small Software Company. *29th International Conference on Software Engineering* (pp. 602-611). Grenoble: IEEE Computer Society.

151. Martin, R. (2003). *Agile Software Development: Principles, Patterns and Practices.* Upper Saddle River: Prentice Hall.

152. Mathiassen, L., & Stage, J. (1992). The principle of Limited Reduction in Software Design (Information, Technology and People). *Information, Technology and People*, pp.171-185 (Vol.6:No2/3).

153. Mattiello-Francisco, F., Martins, E., Cavalli, A. R., & Yano, E.-T. (2011). An approach for testing interoperability and robustness of real-time embedded software. *Journal of Systems and Software*, pp.3-15 (Vol.85;No.1).

154. May, & Zimmer. (1996). The Evolutionary Development Model for Software. *Hewlitt Packard Journal Online*, pp.1-8 (Article 4).

155. McClelland, D. (1976). *A guide to job competency assessment.* Boston: McBer and Company.

156. McClelland, D., Atkinson, J., Clark, R., & Lowell, E. (1953). *The achievement motive.* New York: Appleton-Century-Crofts.

157. McCracken, D., & Jackson, M. (1982). Life Cycle Concept Considered Harmful. *ACM SIGSOFT Software Engineering notes*, pp.27-32 (Vol.7;No. 2).

158. McFarlan, F. W. (1981). Portfolio approach to information systems. *Harvard Business Review*, pp.142-150 (Issue:September-October).

159. McGrath, J. (1984). *Groups: Interaction and Performance.* Englewood Cliffs: Prentice-Hall.

160. McKeen, J., Guimaraes, T., & and Wetherbe, J. (1994). The relationship between user participation and user satisfaction: An investigation of four contingency factor. *MIS Quarterly*, pp.427-451 (Vol.18;No.4).

161. Meyer, M., & Curley, K. (1991). An applied framework for classifying the complexity of knowledge-based systems. *MIS Quarterly*, pp.455-472 (Vol.15;No.4).

162. Miller, S. D., DeCarlo, R., Mathur, A., & Cangussu, J. (2006). A Control-theoretic approach to the management of a software system test phase. *The Journal of Systems and Software*, pp.1486-1503 (Vol.79).

163. Mills, H., Dyer, M., & Linger, R. (1987). Cleanroom software engineering. *IEEE Software*, pp. (19-25) (Vol. 4).

164. Mitchell, S., & Seaman, C. (2009). A Comparison of Software Cost, Duration, and Quality for Waterfall vs.Iterative and Incremental Development: A Systematic Review. *Third International Symposiumm on Empirical Software Engineering and Measurement* (pp. 511-515). Institute of Electrical and Electronics Engineers.

165. Moe, N., A.B., A., & Dybå, T. (2012). Challenges of shared decision-making: A multiple case study of agile Software Development. *Information and Software Technology*, pp 853-865 (Vol.54).

166. Muccini, H., Dias, M., & Richardson, D. (2006). Software architecture-based regression testing. *The journal of systems and software*, pp.1379-1396 (Vol.79).

167. Mumford, E. (1983). *Designing Human Systems.* Manchester Business School.

168. Munassar, N., & Govardhan, A. (2010). A Comparison Between Five Models Of Software Engineering. *International Journal of Computer Science Issues*, pp.94-101 (Vol. 7, Issue 5).

169. Murphy, G., & Salomone, S. (2013). Using social media to facilitate knowledge transfer in complex engineering environments : a primer for educators. *European Journal of Engineering Education*, pp. (70-84) (Vol.38; No.1).

170. Myers, G. (1979). *The Art of Software Testing.* John Wiley and Sons.

171. Myers, M., & Klein, H. (2011). A set of Principles for Conducting Critical Research in Information Systems. *MIS Quarterly*, pp.(17-36) (Vol.35; No.1).

172. Naur, P. (1985). Intuition in Software Development. *Lecture Notes in Computer Science*, pp. 60-79 (Vol. 186).

173. Neisser, U. (1976). General, academic and artificial intelligence . In L. Resnick, *The nature of intelligence* (pp. 135-144). Hillsdale, NJ: Erlbaum.

174. Nerur, S., Mahapatra, R., & Mangalaraj, G. (2005). Challenges of Migrating to AGile Methodologies. *Communications of the ACM*, pp.73-78 (Vol. 48; No. 5).

175. Nguyen, N., Taylor, J., & Bradley, S. (2003). *Job autonomy and job satisfaction: new evidence.* Lancaster University Management School.

176. Nidhraa, S., Yanamadalaa, M., Afzalb, W., & Torkara, R. (2013). Knowledge transfer challenges and mitigation strategies in global software development—A systematic literature review and industrial validation. *International Journal of Information Management*, pp. (333– 355) (Vol.33; No.2).

177. Nonaka, I., & Takeuchi, H. (1995). *The Knowledge-Creating Company*. Oxford University Press.

178. Nonaka, I., & Von Krogh, G. (2009). tacit Knowledge and Knowledge Conversion: Cotroversy and Advancement in organizational Knowledge Creation Theory. *Organization Science*, pp(635-652) (Vol. 20; No. 3).

179. O'Brien, R. M. (2007). A Caution regarding Rules of Thumb for Variance Inflation Factors. *Qualty & Quantity*, pp. 673-690 (Vol. 41; No. 5).

180. Orilkowski, W. (1993). CASE Tools as Organizational Investigating Incremental and Radical Chages in Systems Development. *MIS Quarterly*, pp.309-340.

181. Paetsch, F., Eberlein, D. A., & Maurer, D. F. (2003). Requirements Engineering and Agile Software Development. *Proceedings of the Twelfth IEEE International Workshop on Enabling Technologies* (pp. 308-313). IEEE Computer Society.

182. Palmer, S., & Felsing, J. (2002). A Practical Guide to Feature-Driven Development. Englewood cliffs, New Jersey: Prentice Hall.

183. Patel, C., & Ramachandran, M. (2008). Bridging Best Traditional SWD Practices with XP to Improve the Quality of XP Projects. *International Symposium on Computer Science and its Applications* (pp. 357-360). IEEE Computer Society.

184. Pee, L., Kankanhalli, A., & Kim, H. (2010). Knowledge Sharing in Information Systems Development: A Social Interdependence Perspective. *Journal of the Association for Information Systems*, pp.(550-575) (Vol.11; No.10).

185. Perkusich, M., Soares, G., Almeida, H., & Perkusich, A. (2015). A procedure to detect problems of processes in software development using Bayesian networks. *Expert Systems with Applications*, pp. (436-450) (Vol.47).

186. Perrow, C. (1984). *Normal Accidents Living With High-Risk Technologies.* New York: Basic Books Inc.

187. Petersen, K., & Wohlin, C. (2010). The effect of moving from a plan-drivento an incremental software developmentapproach with agile practices. *Empirical Software Engineering*, pp 654-693 (Vol.15).

188. Pfleeger, S. (2001). *Software Engineering.* Upper Saddle River: Prentice Hall.

189. Polanyi, M. (1962). *Personal Knowledge.* Chicago: The University of Chicago Press.

190. Polanyi, M. (1966). *The Tacit Dimension.* Chicago: University of Chicago Press.

191. Rabelo, J., Oliveira, E., Viana, D., Braga, L., Santos, G., Steinmacher, I., et al. (2015). Knowledge Management and Organizational Culture in a Software

Organization - a Case Study. *IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering*, pp 89-92.

192. Rajagopalan, S. (2014). Review of the Myths on Original Software Development Model. *International Journal of Software Engineering & Application*, pp (103-111) (Vol. 5; No. 6).

193. Ramesh, B., Cao, L., Mohan, K., & Xu, P. (2006). Can distributed sofwtare development be agile. *Coomunications of the ACM*, pp.41-46 (Vol. 49, No. 10).

194. Ribbers, P., & Schoo, K. (2002). Program Management and complexity of ERP implementation. *Engineering Management Journal*, pp.42-52 (Vol.14;No.2).

195. Ribeiro, R. H. (2007). The bread-making machine: Tacit knowledge and two types of action. *Organisational Studies*, pp.(257–262) (vol. 28; no. 9).

196. Ringle, C., Wende, S., & Will, S. (2005). SmartPLS 2.0 (M3) Beta. [Online]. Available:http://www.smartpls.de.

197. Rothermal, G., & Harrold, M. (1993). A safe efficient algorithm for regression test selection. *Proceedings of the conferece on software maintenance* (pp. 358-367). CSM.

198. Rothermal, G., & Harrold, M. (1996). Analyzing regression test selection techniques. *Software Engineering . IEEE Transactions*, pp.529-551.

199. Royce, W. (1970). Managing the Development of Large Software Systems.

200. Rumbaugh, J., & Jacobson, I. (1999). *The Unified Software Development Process.* MA: Addison-Welsey.

201. Rus.I, Lindvall.M, & Sinha, S. (2001). *Knowledge Management in Software Engineering.* Maryland: Fraunhofer Cente for Experimental Software Engineering.

202. Ryan, S., & O'Connor, R. V. (2009). Development of a team measure for tacit knowledge in software development teams. *Journal of Systems and Software*, pp.229-240 (Vol. 82; No. 2).

203. Sabberwal, R., & Elam, J. (1996). Overcoming Problems in Information Systems Development By Building and Sustaining Commitment. *Accounting, Management and Information technologies*, pp.283-309.

204. Sabberwal, R., & Robey, D. (1995). Reconciling Variance and Process Strategies for Studying Information Systems Development. *Information Systems Res*, pp.303-323.

205. Schüler, J., Sheldon, & Frolich, S. M. (2010). Implicit need for achievement moderates the relationship between competence need satisfaction and subsequent motivation. *Journal of Research in Personality*, pp.1-12.

206. Schwaber, K., & Beedle, M. (2001). Agile Software Development with Scrum. Englewood Cliffs, New Jersey: Prentice Hall.

207. Senguta, B., Chandra, S., & Sinha, V. (2006). A Research Agenda for Distributed Software Development. *Proceedings of the 28th international conference on software engineering* (pp. 731-740). New York: ACM.

208. Shenhar, A., & Dvir, J. (1996). Toward a typological theory of project management. *Research Policy*, pp.607-632.

209. Šmite, D., Wohlin, C., Gorschek, T., & Feldt, R. (2010). Empirical evidence in global software engineering:A Systemic Review. *Empirical Software Engineering* , pp. (91-118) (Vol.15;No.1).

210. Sommerville, I. (2007). *Software Engineering 8.* Addison-Wesley.

211. Sommerville, I., Cliff.D., Calinescu, R., Keen, J., Kelly, T., Kwiatkowska, M., et al. (2012). Large-scale Complex IT Systems. *Communications of the ACM* , pp. 71-77 (Vol.55;No.7) .

212. Sorensen, C. (1993). Adoption of CASE tools - An Empirical Investigation - PhD Thesis. *Department of Mathematics and Computer Science, University of Aalborg, Denmark*.

213. Staats, B., Valentine, M., & Edmondson, A. (2010). Using What We Know: Turning Organizational Knowledge into Team Performance. *Harvard Business School*, pp.(1-35).

214. Standish Inc., T. S. (2009). CHAOS SUmmary 2009: The 10 laws of Chaos.

215. Stapleton, J. (1997). DSDM Dynamic Systems Development Method. Addison-Wesley.

216. Sternberg, R., Forsythe, G., Hedlund, J., Horvarth, J., Wagner, R., Williams, W., et al. (2000). *Practical Intelligence in everyday life.* Cambridge University Press.

217. Sternberg, R., Forsythe, G., Hedlund, J., Horvath, J., Tremble, T., Williams, W., et al. (1999). *Tacit Knowledge in the Workplace.* Alexandria: U.S. Army Research Institute for the Behavioral and Social Sciences.

218. Strauss, A., & Corbin, J. (1990). *Basics of Qualitative Research: Grounded Theory Procedures and Techniques.* Newbury Park, CA.: Sage Publications.

219. Striebeck, M. (2005). Ongoing Quality Improvement, or: How We All Learned To Trust XP. *Proceedings of the Agile Development Conference* (pp. 267-271). Denver: IEEE Computer Society.

220. Sun, R., & Merill, R. (2001). From Explicit Skills to Implicit Learning: A Bottom-Up Model of Skill Learning . *Cognitive Science*, pp203-244 (Vol. 25; No. 2).

221. Tait, P., & Vessey, I. (1988). The effect of user involvement on system success: A contingency approach. *MIS Quarterly*, pp.91-108 (Vol.12;No.1).

222. Talby, D., Karen, A., Hazzan, O., & Dubinsky, Y. (2006). Agile Software Testing in a Large-Scale Project. *IEEE Software*, pp.30-37.

223. Tarha, A., & Yilmaz, S. (2014). Systematic analyses and comparison of development performance and product quality of Incremental Process and Agile Process. *Information and Software Technology*, pp. (477-494) (Vol.56).

224. Thorpe, R., & Homan, G. (2000). *Strategic Reward Systems.* Prentice Hall.

225. Tsoukas, H. (2002). Do we really understand tacit knowledge? In M. Easterby-Smith, & M. Lyles, *Handbook of Organizational Knowledge Management* (pp. pp. (410–27)). Blackwell .

226. Tsui, F., & Iriele, S. (2011). Analysis of Software Cohesion Attribute and Test Case Development Complexity. *Proceedings of the 49th Annual Southeast Regional Conference* (pp. pp.(237-242)). ACM.

227. Tsui, F., & Karam, O. (2007). *Essentials of software engineering.* Ontario: Jones and Bartlett.

228. Turk, al., D., France, R., & Rumpe, B. (2000). Limitations of Agile Software Processes. *Third International Conference on eXtreme Programming and Agile Processes in Software Engineering* (pp. 43-46). Springer-Verlag.

229. Turk, D., France, R., & Rumpe, B. (2005). Assumptions Under lying Agile Software Development Processes. *Journal of Database Management*, pp. 62-87 (Vol.16;No.4).

230. Turner, J., & Cochrane, R. (1993). Goals-and-methods matrix: Coping with projects with ill defined goals and/or methods of achieving them. *International Journal of Project Management*, 93-102.

231. Urquhart, C., Lehmann, H., & Myers, <. (2010). Putting the 'theory' back into grounded theory: guidelines for grounded theory studies in information systems. *Information Systems Journal*, 357-381 (Vol.20;No.4).

232. Venkatesh, V., Brown, S., & Bala, H. (2013). Bridging the Qualitative–Quantitative Divide:Guidelines for Conducting Mixed Methods Research in Information Systems. *MIS Quarterly*, pp.(21-54)(Vol.37;No.1).

233. Vidgen, R., & Madsen, S. (2003). Exploring the socio-technical dimension of information systems development: use cases and job satisfaction.

234. Von Krogh, G. (2012). How does Social Software Change Knowledge Management? Toward a Srategic Research Agenda. *Journal of Strategic Information Systems*, pp 154-164 (Vol. 21).

235. Wagner, K., & Sternberg, R. (1985). Practical intelligence in real-world pursuits: The role of tacit knowledge. *Journal of personality and social proceedings*, pp.436-458, (Vol. 49, no. 2).

236. Wallace, L., & Keil, M. (2004). Software project risks and their effect on outcomes. *Communications of the ASM*, pp.68-73 (Vol. 47, No. 4;).

237. Walter, T., & Grabowski, J. (1999). A framework for the specification of test cases for real-time distributed systems. *Information and Software Technology*, pp.781-798.

238. Wang, M., Huang, C., & Yang, T. (2012). The Effect of the Project Environment on the Relationship between Knowledge Sharing and Team Creativity in the Software Development Context. *International Journal of Business and Information*, pp. (59-80) (Vol.7; No. 1).

239. Wegener, J., Baresel, A., & Sthamer, H. (2001). Evolutionary test environment for automatic structural testing. *Information and Software Technology*, pp.841-854 (Vol.43; No.14).

240. Willcocks, L., & Margetts, H. (1994). *Risk and Information Systems: Developing the Analysis.* London: Chapman-Hall.

241. Williams, T. (1999). The need for new paradigmsfor complex projects. *International journal for project management*, pp. 269-273.

242. Wong, W., Horgan, J., London, S., & Mathur, A. (1998). Effect of test set minimization on fault detection effectiveness. In *Software - Practice and Experience* (pp. 347-369).

243. Wood, R. (1986). Task Complexity: Definition of the construct. *Organizational Behavior and Human Decision Processes, 37,* pp.60-82.

244. Woodside, A. (2010). Bridging the chasm between survey and case study research: Research methods forachieving generalization, accuracy, and complexity. *Industrial Marketing Management*, pp. (64-75) (Vol. 39).

245. Woodside, A., & Baxter, R. (2013). Achieving accuracy, generalization-to-contexts, and complexity in theories of business-to-business decision processes. *Industrial Marketing Management*, pp. (382-393); (Vol. 42).

246. Xia, W., & Lee, G. (2005). Complexity of Information Systems Development Projects: Conceptualization and Measurement Development. *Journal of Management Information Systems*, pp.45-82 (Vol.22; No.1;).

247. Yeates, D., Shields, M., & Helmy, D. (1994). *Systems analysis and design*. Pittman publishing.

248. Yin, R., & Ding, X. (2012). How to Improve the Quality of Software Testing. *2012 International Conference on Systems and Informatics* (pp. pp.(2533-2536)). Yantai : ICSAI.

249. Yoo, S., & Harman, M. (2010). Regression Testing Minimisation, Selection and Prioritisation : A Survey. *Software Testing, Verification and Reliability*, pp. 67-120 (Vol.22; No.2).

250. Zack, Z., McKeen, J., & Singh, S. (2009). Knowledge management and organizational performance: an exploratory analysis. *Journal Of Knowledge management*, pp.392-409 (Vol.13;No.6).

251. Zheng, M., Alager, V., & Ormandjieva, O. (2008). Automated generation of test suites from formal specifications of real-time reactive systems. *The journal of systems and software*, pp.286-304 (Vol. 81).

8 Appendix

8.1 Analysis of Qualitative Data

Comments are coded in terms of the count of similarly expressed sentiments. The counts identify whether they can be attributed to a participant with < 10 years' experience (red, (inexperienced or i)) or a participant with >= 10 years' experience (blue, (experienced or e)), e.g. the following comment relating to the importance of tacit knowledge:

The system under test is described as consisting of a significant amount of complex features. Even though a considerable amount of explicit knowledge relating to the system is freely available, it has been stated that a good deal of knowledge relating to the system, which is demanded for effective system testing, is actually tacit in nature. (20:i) (17:e). This breaks down as being associated with the following methodologies: traditional (25); agile (12).

The level of documentation does help reduce complexity but enterprise systems are described as often being very complex with only a certain amount of such knowledge lending itself to being made explicit and documented (27:i) (12:e). (This breaks down as traditional: (25); agile: (14). A significant amount of time is spent trying to acquire tacit knowledge from development, especially in relation to system interactions and expected outcomes under different conditions (14:i) (11:e). This breaks down as traditional: (18); agile: (7) If the knowledge is not freely flowing then this can make the process a lot more inefficient and complex (2:i) (6:e).

8.1.1 Initial Coding of Complexity Associated with the System under Test

Classification	***Statement***
The impact of System Complexity, primarily relating to	System test planning i.e. deciding what aspects of the system can and should be tested can be a complex activity (28:i) (13:e), often due to system interoperability and interdependencies associated with different elements of the system (8:i) (9:e). The system is complex, with the number of different configurations applying to

System test planning	system deployment. Complexity is embodied in the product. Directory structure knowledge (knowledge of who knows what) was mentioned as being very important regarding system use, and how the system can be used under certain circumstances. This was something which is described as difficult to make explicit (3:i). There needs to be a complete understanding of how the feature/system is expected to operate, and how it could be used (8:i) (12:e). The view was expressed that complexity at the planning stage can affect one's ability to specify appropriate tests, which can have a knock-on effect on the system test development stage (5:i) (4:e), and can contribute to complexity associated with the test execution stage (9:i) (2:e). This can also affect the ability to debug the system at the fault analysis stage, and the test measurement stage. A lack of system understanding can influence one's ability to carry out estimation of required test resources i.e. human, technical and time (4:i) (2:e).
The impact of System Complexity, primarily relating to System test development	The implementation of test cases as planned, an activity which must be carried as part of the test development stage, can be quite a complex task, (21:i) (11:e). Some refer to this as being due to the interoperability and interdependencies associated with different elements of the system (4:i) (4:e). If an effective test environment is not implemented, this is described as causing trouble for later stages of system testing (5:i) (2:e).
The impact of System Complexity, primarily relating to System test execution	General reference was made to complexity associated with system test execution stage of system testing i.e.: *If tests have not been specified properly or clearly defined, then it can introduce complexity at the test execution stage (3:i) (9:e), particularly if testing is manual in nature, as opposed to being automated (8:i) (5:e), with non-standard or exploratory testing being carried out (2:i).* The involvement of complexity relating to the system under test was not explicitly mentioned, but it cannot be ruled out, particularly in the case of a manual testing approach being

	adopted.
The impact of System Complexity, primarily relating to System test fault analysis	System complexity affects the ability to carry our fault analysis or debug on potential issues, and to be able to differentiate between what is an actual bug, and what is a test environment issue. The fault analysis stage demands an understanding of the exact test which was being performed i.e. what the test was attempting to achieve, what effect it had on the system, and what effect it should have had on the system (17:i) (16:e).
The impact of System Complexity, primarily relating to System test measurement	General reference was made to complexity associated with system test measurement stage of system testing i.e.: *Automation of test case measurement can remove complexity, but complexity appears to come into play when deeper analysis is carried out as part of the test measurement stage, in order to accurately evaluate the quality of the system under test (4:i) (3:e). A balance must be achieved between adequate system quality against time to market pressures (1:i) (1:e).* Even though complexity relating the system under test was not explicitly mentioned, this cannot be ruled out as being a source of complexity, especially concerning the manual assessment of system quality.
The impact of System Complexity, primarily relating to System test management	Therefore test management was not found to be impacted by complexity associated with the system under test.

8.1.2 Initial Coding of Tacit Knowledge relating to the System under Test

Highlighted below is evidence of tacit knowledge relating to the system under test, as it impacts the various stages of system testing. Tacit knowledge was distinguished

from explicit knowledge in that it was described in terms of knowledge which was difficult to articulate and was acquired through experience. Evidence of such knowledge was found in the case of system test planning, test case development, which is carried out as part of test case development, and test debug, which happening as part of the system test fault analysis stage.

Test planning related tacit knowledge associated with the system under test	As previously stated, the availability of tacit knowledge relating to the system under test is essential to enabling effective completion of the planning stage (27:i) (14:e). Enterprise projects generally require a considerable amount of system knowledge right through the test planning and test development stages (5:i) (5:e). This may not be as easy to acquire if the system being implemented is a bespoke system, being developed from scratch by a separate team (3:i) (1:e), or in the case of a geographically dispersed test team (1:i).
Test development related tacit knowledge associated with the system under test	Views were expressed that projects generally require a considerable amount of system related tacit knowledge throughout the test development stage (15:i) (7:e). The availability of such tacit knowledge relating to the system under test, interoperability etc. is imperative to successfully completing the test development stage (15:i) (10:e). Test environments must accurately reflect the final deployment scenario at customer sites. For test case development, and to enable effective assessment of automation possibilities, there needs to be an understanding of what has to be tested and how the system could eventually be used (2:i) (2:e).
Test execution related tacit knowledge associated with the system under test	The effect of tacit knowledge associated with system test execution, which relates to the system under test, is not something which was explicitly mentioned. It was stated that a certain amount of test execution related knowledge does lend itself to being made explicit, but numerous other participants did mention that there was a relationship between manual testing, and tacit knowledge i.e.:

	A certain amount of the test execution normally lends itself to be made explicit (15:i) (6:e). Others stated that test suite execution had a strong relationship to tacit knowledge (10:i) (6:e), but such views related to when manual approaches to testing were adopted, involving complex test steps, such as load testing, or exploratory testing, requiring a more detailed knowledge (2:i) (3:e). A relationship between at least some aspects of system test execution, such as load, or stress testing, and tacit knowledge, cannot be ruled out.
Fault analysis related tacit knowledge associated with the system under test	To fully appreciate what component of the system bugs are emanating from, one requires tacit knowledge relating to the system under test and how it interoperates (16:i) (8:e). Contrasting with a common expressed view, some expressed the view that a lot of debug knowledge can be made explicit (2:i) (1:e). Debugging of issues often brings a dependency on system development teams for applicable knowledge (8:i) (9:e), or other team members (including those focussed on development and maintenance of the test environment) (2:i) (4:e).
Test measurement related tacit knowledge associated with the system under test	Reference has been made to tacit associated with test case measurement stage of system testing i.e.: *Test case measurement is described as being based on experience (2:i) (3:e), but something with a weak relationship to tacit knowledge (15:i) (10:e). Required tacit knowledge is associated with current system evaluation against expected, with a balance having to be achieved between available test resources, and the achievement of sufficient quality of the system within a certain timeframe (6:i) (3:e). Test case measurement can be taken care of, to a large extent, on an automated basis, which simplifies matters (3:i) (3:e), more or less consisting of a recording of a pass or fail after test execution (4:i) (1:e).*

	Although there were no direct references made regarding the relationship between test case measurement and tacit knowledge associated with the system under test, it was mentioned that required tacit knowledge does come into play with the evaluation of system quality against expected. It therefore could not be ruled out that at least some of this knowledge relates to understanding of the system under test.
Test management related tacit knowledge associated with the system under test	Reference has also been made to tacit associated with the test case management stage of system testing i.e.: *Test case management was expressed to have a moderate dependence on tacit knowledge i.e. it does lend itself to being made explicit (10:i) (10:e). However, the introduction of new systems, modifications to test environments, or optimisation efforts, can make test environments quite complex to manage, with some dependence on tacit knowledge (6:i) (1:e).* As highlighted above, evidence was found regarding test management, but such knowledge was found to relate to the test environment and the wider process of system testing, as opposed to the system under test.

8.1.3 Initial Coding of Complexity associated with the Process of System Testing

An effort was also made to highlight complexity which affects the various stages of system testing but is not directly associated with the system under test, or where the knowledge may be related to the system under test but explicit in nature, such as in the case of functional specifications.

Complexity associated	Functional requirement specifications which have been poorly specified can be a contributor to complexity associated with the test

with the task of system test planning	planning stage (10:i) (5:e). The exposure of testers to requirement details at a late stage in the development process, can also impact the effectiveness and efficiency of the system tester to plan effective system tests (1:i) (1:e). A lack of information available at the planning stage, specifications etc. leads to a deficit of knowledge, which can introduce a lot of complexity at this and later stages (7:i) (4:e). Conversely good planning makes the subsequent development stage less complex (5:i) (1:e). Decisions to be made at the planning stage, such as those relating to what exactly is feasible in terms of meeting system test requirements with available resources e.g. test case selection and prioritisation of test cases, can also introduce complexity for planning and later stages of testing (12:i) (4:e). A balance must be achieved between adequate system quality and time to market pressure (11:i) (7:e). Achievement of such a balance can have a direct impact on the prioritisation and selection of test cases. An initial risk assessment, carried out as part of the test case planning stage, detailing what can and should be tested, within a certain period of time, is described as being very complex. Complexity can come into play when broader product knowledge or customer deployment knowledge is not readily available (3:i) (1:e).
Complexity associated with the task of system test development	To build a test environment which is reflective of final deployment can also be a quite complex process (10:i) (5:e). There is often a deficit of standards or guidance regarding set up of test environments which accurately reflect customer deployments, and often insufficient knowledge relating to the actual system in practice (11:i) (9:e).Test configuration can be very complex to set up, especially for somebody of lesser experience (1:i) (1:e). The role of system test automation is cited as a potential contributing factor to complexity (13:i) (4:e), with the development of such automated systems described as often being a complex process (1:i) (3:e). Sometimes automation is insisted, even though it may not be an appropriate fit i.e. it may not be possible to transfer the manual tests, to an automated platform, while still retaining the ability to effectively test the desired operational characteristics of the system (7:i) (4:e). Although not necessarily complex, there can be time to

	market and cost pressures associated with setting up automated environments (1:i) (1:e). There are many different routes for successful testing to be achieved and this can also introduce complexity (2:i) (5:e). Development teams often set acceptance criteria for system tests (1:i), with varying levels of detail involved. The exposure of testers to the introduction or modification of feature details at a late stage in the development process, often impacts the effectiveness and efficiency of the system tester to develop effective system tests (4:i) (1:e).
Complexity associated with the task of system test execution	If tests have not been specified properly or clearly defined, then it can introduce complexity at the test execution stage (8:i) (9:e), particularly if testing is manual in nature, as opposed to being automated (8:i) (5:e), with non-standard or exploratory testing being carried out (2:i). A lot of this knowledge can be made explicit (5:i) (7:e), but this isn't necessarily always done (1:e). One tends to go into more detail in tests and test steps with more experience. With additional detail comes additional complexity (3:e). Such additional detail often has a strong link to tacit knowledge. Test suite execution, does benefit from effective work which has been carried out at the planning and development stages, but there can be complexity emanating from requirement changes which may surface during test suite execution, particularly if the execution is manual in nature (6:i) (1:e).
Complexity associated with the task of system test fault analysis	When carrying out fault analysis, one needs to rule out the involvement of the test environment, as opposed to the system under test (13:i) (6:e). Debugging can prove to be complex (35:i) (17:e), with a certain dependency on the experience of the tester (10:i) (6:e), and on development teams (8:i) (6:e). Often this can be quite time consuming (days in some instances) and at the same time you are under pressure to finish your tests (2:i). A bug in one component could cause a bug in another component and this must be understood and be identifiable (1:i) (1:e). The automation of test cases is described as something which contributes greatly to general complexity associated with system testing. Sometimes automation masks the exact system interoperability, thereby having the effect of

	reducing the general understanding of system operation (3:i) (1:e).
Complexity associated with the task of system test measurement	Test case measurement can be taken care of, to a large extent, on an automated basis or by a separate team, so may be relatively simplistic (13:i) (8:e), and more or less consisting of recording a pass or fail after test execution (10:i) (5:e). Automation of test case measurement can indeed remove complexity, but complexity appears to come into play when deeper analysis is carried out as part of the test measurement stage, in order to accurately evaluate the quality of the system under test (8:i) (6:e). A balance must be achieved between adequate system qualities against time to market pressures (1:i) (2:e). Customer deployed environments are described as being significantly more complex and larger than the test environments which are available to system test, and therefore there is always an offset which one must be aware of regarding the evaluation of quality (1:i). It can be hard to determine whether all resources are being maximised and whether testing is being carried out in line with customer deployment as well as possible. A heavily automated system with no automated quality measurement framework built in, is described as contributing to such complexity (1:i). The inclusion of aspects of code quality such as code coverage, may also contribute to complexity associated with quality measurement. Complex measurement frameworks, which must be approached on a manual basis, can prove quite challenging, especially when aspects of quality such as code coverage, are considered as part of quality evaluation (2:i).
Complexity associated with the task of system test management	Management of resources, which involves the balancing of resources associated with the test environment, and enabling test case preservation, can be quite a complex task (14:i) (8:e). Such management is stated as requiring experience and know-how in order to balance resources properly (1:i). Most of this knowledge can be made explicit (2:i) (2:e).This would relate to getting people on board and trying to speed up the process of getting necessary resources, so it is described as being more difficult (time consuming), than complex. The management of fix testing can be quite a complex task, with pressure for fix signoff. Test environment

	changes, in terms of the analysis and consideration of changes, can introduce complexity (1:i) (4:e). If substantial architectural changes occur during the test process, this can prove complex to manage, particularly if major test environment changes are necessary (2:i) (1:e). Such changes can affect all stages of system testing (1:i).

8.1.4 Initial Coding of Tacit Knowledge related to the Process of System Testing

The following section highlights evidence of tacit knowledge relating tacit knowledge which affects the various stages of system testing but is not directly associated with the system under test. Tacit knowledge was distinguished from explicit knowledge in that it was described in terms of knowledge which was difficult to articulate and was acquired through experience. Evidence of such knowledge was found in the case of system test planning, the test environment, which is carried out as part of test case development, test case execution, and test debug, which happens as part of the system test fault analysis stage.

Tacit knowledge relating to the task of system test planning	The importance of tacit knowledge relating to test case planning, which is gained through experience, was emphasised by numerous participants (16:i) (11:e). However the view was expressed by a smaller number of participants that a certain amount of planning related knowledge, specifications etc. and can be made explicit, thereby reducing the dependency on tacit knowledge (8:i) (1:e). A shortfall in tacit knowledge could result in a lack of appreciation for what tests are necessary in order to test the system properly, given available resources (2:i) (5:e), is something which has a strong influence on the final quality of the system. This has been described as an issue one needs to be conscious of, particularly in the case of testing being outsourced, and a limited access to appropriate tacit knowledge. Criteria which are used to determine the quality of the system may have been set by either the system implementer or the eventual customer (possibly set by project manager or system architect) (1:i) (1:e). Sometimes there may be detail you may be missing during the planning stage, detail which may only become apparent with an understanding and experience of both system

	testing, and the actual system under test (2:i) (4:e).
Tacit knowledge relating to the task of system test development	Applicable test environment development knowledge is usually tacit in nature and difficult to make explicit (15:i) (13:e). Knowledge relating to the test environment is usually acquired through experience (4:i) (3:e), and may not be as easy to acquire if the system being implemented is a bespoke system, being developed from scratch by a separate team (3:i) (1:e), or in the case of a geographically dispersed test team (1:i). A contrary view was expressed by some, that a lot of test environment knowledge can actually be made explicit (1:i) (2:e).
Tacit knowledge relating to the task of system test execution	A certain amount of the test execution normally lends itself to be made explicit (15:i) (6:e). Others stated that test suite execution had a strong relationship to tacit knowledge (6:i) (10:e), but such views related to circumstances when manual approaches to testing were adopted, involving complex test steps, such as load testing, or exploratory testing, requiring more detailed test environment knowledge (3:i) (2:e).
Tacit knowledge relating to the task of fault analysis	When carrying out fault analysis, one needs to rule out the involvement of the test environment, as opposed to the system under test (13:i) (6:e), such ability is primarily dependent on the experience of the tester (10:i) (6:e). Debug can often be quite time consuming (days in some instances) and at the same time you are under pressure to finish your tests (2:i). If there are delays in delivery of an appropriate response from development or test environment focussed/automation teams, this can elongate the test process and have a knock-on effect on issue resolution. Directory structure associated with who to talk to under what circumstances is described as something which have an impact on the fault analysis stage of system testing (11:i) (1:e). Complex test steps can make fault analysis more complex (3:i). Knowledge associated with automation is described as being primarily tacit in nature (3:i). Debugging of issues often brings a dependency on other team members (including those focussed on the development and maintenance of the test environment) (2:i) (4:e).

Tacit knowledge relating to the task of system test measurement	Test case measurement is described as being based on experience (2:i) (3:e), but something with a weak relationship to tacit knowledge (15:i) (10:e). Required tacit knowledge is associated with current system evaluation against expected, with a balance having to be achieved between available test resources, and the achievement of a sufficient level of system quality, within a certain timeframe (6:i) (3:e). Test case measurement can be taken care of, to a large extent, on an automated basis, which simplifies matters (3:i) (3:e). This can simply consist of a recording of a pass or fail after test execution (4:i) (1:e).
Tacit knowledge relating to the task of system test management	Test case management was expressed to have a moderate dependence on tacit knowledge i.e. it does lend itself to being made explicit (10:i) (10:e). However, the introduction of new systems, modifications to test environments, or optimisation efforts, can make test environments quite complex to manage, with some dependence on tacit knowledge (6:i) (1:e).

Table 8.1: Research Data Relating to Tacit Knowledge Associated with the Process of System Testing.

8.2 Recommended Actions to Reduce the Effects of System Test Complexity

8.2.1 The Availability of Tacit Knowledge within the Test Team

Table 5.11, focusses on the importance of system test team members in reducing the effects of system test complexity.

The dependence on knowledge from team members and the availability of SMEs	It has been explained that the availability of subject matter experts (SMEs), providing necessary tacit knowledge relating to the actual system under test or the system test environment, is important in the reduction of complexity (27:i) (16:e). The level of tacit knowledge is described as being proportional to the complexity of the project (1:e). As a general rule, the bigger the

	system test team, the greater the necessity for subject matter experts (SMEs) to be made available to system testers (2:e). This is often a case of sharing the load in terms of knowledge resources. A dedicated SME has been described as helping to provide an ongoing source of tacit knowledge to the system test team for the system test stages of *planning*, *development*, *execution* and *fault analysis* stages (5:i) (3:e). Such an SME relating to the system, may not always be freely available (1:i), and this appears to be particularly prevalent in the case of smaller teams or new enterprise level projects. The lack of availability of an SME often leads to learning on the job, something which can be more difficult for less experienced engineers. An SME, with knowledge pertaining to interacting features, can be difficult to source without involving software developers. Geographically distributed test teams can introduce complexity for system testing (1:i). This is described as being particular pertinent in the case of shared test environments (1:i). The involvement of someone who is familiar with how the system is intended to work in practice i.e. in accordance with the original architecture is of significant value (5:i) (5:e). Such knowledge enables the system test team to carry out some debug analysis, ensuring the debug process is more efficient, by developers not having to consistently debug test environment issues (1:i).

Table 8.2: The Availability of Tacit Knowledge within the Test Team.

8.2.2 The Availability of Knowledge from Development Teams

The following section highlights the benefit of development team knowledge in the reduction of system test complexity. Such knowledge can come in the form of knowledge when lends itself to being made explicit e.g. functional specifications or user stories etc., or tacit knowledge which cannot be easily made explicit, and is best communicated via personal interaction.

The importance of explicit knowledge in reducing complexity	At the test planning stage, there is a great deal of information which can be made available via function specifications, which can help in reducing complexity associated with system testing (33:i) (13:e). The benefit of specifications in reducing complexity associated with system testing is diminished if the functional specifications are incomplete, subject to change, or arrive late in the software development process (1:i) (3:e). Applications which allow the management of formal specifications, and can be used to document aspects of the system, help in providing a better understanding of the original drivers for a particular feature. This information combines with user stories (or specifications), to provide the basis for test case planning, inclusive of acceptance tests. Information regarding system operation which is expected, or not expected, under different operational circumstances, could reduce complexity associated with the planning and development stages (1:e).
The importance of the transfer of tacit knowledge from developers in reducing system test complexity	The level of documentation associated with a development project does help reduce complexity but enterprise systems are described as often being very complex with only a certain amount of such knowledge lending itself to being made explicit and documented (27:i) (12:e). A significant amount of time is spent trying to acquire tacit knowledge from developers, especially in relation to system interactions and expected outcomes under different conditions (14:i) (11:e). If the knowledge is not freely flowing then this can make the process a lot more inefficient and complex (2:i) (6:e). A manual approach to system testing can be badly affected by significant changes to requirements during the development process (1:i). It was suggested that a more agile approach to development is very effective in reducing complexity associated with system testing, through the regular encouragement of communication between test and development teams. This is in contrast to teams being involved in a more traditional approach to software development (2:i) (3:e), described as being particularly applicable to development teams operating on a geographically

	distributed basis. Such interaction has been described as being more important than user stories or specifications, which can sometimes be inaccurate or not current (1:i). *What structure do such communications take?* At the planning stage, knowledge relating to the system/feature under test can come through interactions with development, via walkthroughs, specifications, architectural meetings. These approaches to knowledge transfer are described as being very effective in reducing complexity at the planning and development stages (5:i) (5:e). Also suggested was the concept of workshops, as a medium for the knowledge transfer. These can be arranged between the business units, development and test, and hosted by development teams. The purpose of these workshops was to provide a detailed overview on the user stories involved in the forthcoming delivery, enabling testers to plan and develop tests effectively (1:e). The aforementioned approaches can be helpful because for new projects it may not be possible to use previously defined tests. It is possible for developers to specify or outline initial tests, but often this is either not done or is often extremely lacking in detail. Having development sit in with system test, during a test phase, has shown to be a major reducer of complexity at the test execution / fault analysis stages of testing, through facilitating the transfer of knowledge (3:i). The assistance of development teams in triaging issues as part of the fault analysis stage can have a strong impact in reducing complexity associated with this stage, helping to speed up the test process (5:i). Testers should also be encouraged to highlight all potential issues. It was mentioned that the co-location of development teams with test teams can lead to the opportunity of informal communication with development, helping to reduce complexity through the transfer of tacit knowledge (12:i) (3:e). The point has also been made that even in the absence of co-location of development and test, that if there is a good relationship between the two, and they are accessible via the same time zone, that this is also very beneficial in resolving

	issues quicker (1:i).

Table 8.3: The Availability of Knowledge from Development Teams.

8.2.3 The Benefit of Support Applications and Support Teams

The following section provides viewpoints relating to the benefit of support applications and support teams, in reducing complexity associated with system testing.

The use of support teams	There is a benefit of providing supporting application and support teams, which are separate to system testing, but closely aligned. Examples given are project management teams or test environment development and support teams. Deployment knowledge is sometimes difficult to acquire, and this can have a knock on effect on system test planning and system test development. To acquire clear and accurate detail regarding system usage can be a difficult. Project management (or business analysts) are often used to acquire such knowledge, in order to facilitate system testing. Project management can help in recognising necessary test environment system configuration detail (helping to emulate deployment environment), which must be accounted for during the planning stage (3:i) (1:e). Independent, closely aligned, automation teams e.g. teams concentrating on test automation development, can also help in reducing complexity associated with complex test environments, by providing ongoing support for the test frameworks (7:i) (7:e). Automation tends to remove some complexity from the tester, but one must be careful because this can also reduce test environment knowledge for system testers, if they have not been involved in the automation

	process. Being unfamiliar with the test environment can affect a tester's ability to accurately evaluate the effectiveness of tests or to efficiently debug test results. The automation of test case measurement can significantly reduce complexity. The use of automated systems which may be custom built or off the shelf, can help significantly in reducing complexity associated with test case execution and measurement stages of system testing (3:i) (2:e).
The use of project management applications	Project management tools such as JIRA and Confluence were described as helping to clarify what architectural decisions have been made, and the main drivers for these decisions. Tools such as wiki pages are also described as being effective to detail such architectural decisions (2:e). Other applications such as Zepher help simplify test case measurement, and "Quality Centre" can help reduce complexity associated with test case management, aiding the management of the test environment, and test resources. This combined with a minimal amount of architecture changes to the test environment, after initial setup, make test case management an often easy process. It was stated that developers often prefer that communication with system testing be on more of a formal basis. Formal communication is very important but such communication can in itself be complex, depending on the context. Interactions between various system components which may be detailed in a medium such as flow diagrams, for instance, are beneficial, but such a medium is also described as not possibly facilitating the full transfer of knowledge associated with the interactions of a more complex system. A high degree of such knowledge is described as being tacit in nature, such as may be involved in the description of component interactions (1:e).