ASSESSING QUALITY IN SOFTWARE ENGINEERING: A PRAGMATIC APPROACH.

A dissertation
submitted to the department of Computer Science
of the University of Pretoria
in partial fulfilment of the requirements
for the degree of
Magister Scientia (M.Sc)

Daniel Acton, 99030510
July 2013

Abstract

As long as software has been produced, there have been efforts to strive for quality in software products. In order to understand quality in software products, researchers have built models of software quality that rely on metrics in an attempt to provide a quantitative view of software quality. The aim of these models is to provide software producers with the capability to define and evaluate metrics related to quality and use these metrics to improve the quality of the software they produce over time. These models can be quite cumbersome to implement as they require effort and resources to define and evaluate metrics from software projects.

This dissertation aims to build an understanding of quality in software engineering by investigating those concepts core to the field. The basic concepts of the field are described, including quality, metrics and software engineering processes. Three software quality models and four approaches to using metrics to gain insight into quality are discussed with an aim to understanding the apparent strengths and weaknesses of each.

This dissertation proposes a new approach to using metrics to gain insight into software quality. An equation, called the Product Quality Indicator, is proposed and critically assessed, which uses a combination of metrics based on requirements, tests and defects, to provide some insight into quality. Furthermore, a software product, called Metaversion, which relies on the Subversion Software Configuration Management system is presented. This software, which is a reference implementation of the proposed approach, aims to allow for the automatic collection and evaluation of the Product Quality Indicator.

A case study is discussed where the Metaversion system is used and the results of the evaluation of the Product Quality Indicator are compared with the quality of the software as perceived by the testers responsible for testing the software.

Acknowledgments

First and foremost I would like to thank God for blessing me with an enquiring mind. Secondly, I would like to thank my family for allowing me to exercise my enquiring mind. Last, but not least, I would like to thank my supervisors for helping shape my enquiring mind.

Contents

List of Tables

List of Figures

List of Equations

Chapter 1

Introduction

1.1 Problem statement

One of the factors that may influence a consumer's decision to purchase one product or service over another is the difference in quality between those products. If an organisation produces products or provides services that are of low quality, they may not sell many of these products (economic factors like monopolies notwithstanding) and may find themselves struggling to make money and stay in operation.

In the software engineering field, where an example of the product is a software system and an example of the service is a software development service, the importance of quality is no different. Organisations selling software systems or services are also required to provide high quality systems and services to their customers (and for the same reasons).

In traditional manufacturing concerns, there are standards and process models that organisations can adopt and follow that aid them in ensuring their processes are sufficient and appropriate to their concerns. The same is true for software engineering concerns.

In software engineering organisations, however, it appears that there is less focus on implementing and adhering to processes, than actually delivering the projects on time and within budget. There even appears to be an acceptance in some quarters that quality is something that can be traded for delivery within tighter timelines, with quotes such as "I'd rather have it wrong than have it late. We can always fix it later" (Chrissis et al., 2003) being attributed to some software project managers. The reasons for this stem from the reliance on software systems of just about every organisation in business, and their perceived need not to be hindered by the slow implementation (or lesser quality) of these systems.

There are many ways in which software engineering organisations can aid the drive for product quality. Some approaches are process-driven. The idea is that if an organisation follows a "good" process or set of processes, the organisation will consistently deliver the same quality products or deliver the same quality services, and if these processes are maintained and improved over time, the quality of the organisation's outputs will remain steady or increase. In software engineering, processes are used to guide the work effort required to produce a software product or provide software services. The steps of these processes yield outputs that contribute towards the software product or service. It should be borne in mind, though, that focussing on process quality alone may not be sufficient to ensure product quality as discussed by Kitchenham and Pfleeger (1996).

In a software engineering project, measurements and metrics can be used to provide insight into the process, product and project. According to International Organization for Standardization (2005), a process is the "set of interrelated or interacting activities which transforms inputs into outputs", a product is the "result of a process", and a project is the "unique process, consisting of a set of coordinated and controlled activities with start and finish dates, undertaken to achieve an objective conforming to specific requirements, including the constraints of time, cost and resources" (where a requirement is a "need or expectation that is stated, generally implied or obligatory").

Measurements and metrics can be used to deduce, for example, productivity of developers or defects per thousand lines of code. One possible use of measurements and metrics is to deduce information about the quality of the product being produced or service being provided. Metrics and measurements may not cover all of the aspects and complexity of quality in software and the limitations of metrics as a management tool is recognised. Metrics are intended as guidelines, not as absolute indications of quality.

This dissertation aims to investigate the problem that all software producers are faced with which is, that while software quality is very important, it is not trivial to measure.

1.2 Research objectives

The overall aim of this dissertation is to investigate mechanisms to infer information about quality in software, and the objectives are as follows.

- Investigate approaches to measuring quality in Software Engineering projects and identify apparent issues and constraints with these approaches. This is

important to understand the *status quo* of using metrics to infer information about quality.

- Propose a new approach to measuring quality in Software Engineering projects. This approach will attempt to address the issues and constraints of the discussed approaches. The new approach will rely on information and systems that would normally be part of a Software Engineering project, and hence should provide a pragmatic approach to measuring software quality.
- Describe an implementation of this new approach. This aims to show that the approach is applicable to Software Engineering projects and provide an example of how the approach can be practically implemented.
- Using this implementation of the new approach, perform a case study of a real-world Software Engineering project. This case study aims to show the use of the approach and provide some data about how the approach provides insight into quality.
- As part of the case study, a survey is answered by users of the software regarding their perception of the quality of the software produced. These results will be compared with the the results reported by the implementation of the new approach during the case study. This is done to ascertain whether the proposed approach bears any similarity to quality as perceived by users of the software.

1.3 Dissertation structure

This dissertation proceeds in Chapter 2 with a study of the literature surrounding topics pertinent to the field of Software Engineering and measuring quality in software products. In this chapter, the concepts of quality, software engineering and software engineering processes, software requirements, software testing and defects, software quality metrics, and software configuration management will be discussed. Chapter 2 aims to provide insight into software quality and measuring software quality in software engineering projects. In addition, the relationships between process and quality and tests, defects and requirements will be investigated.

Chapter 3 provides a discussion of approaches to measuring quality in software used by other researchers in the field. Each approach is described in detail, and perceived advantages and disadvantages of the approaches are discussed.

Chapter 4 builds on the research discussed in Chapter 3 and the background and context discussed in Chapter 2 to propose a new approach to assessing quality in

software engineering projects. The approach proposes the use of a Software Configuration Management system to store information about the tests, defects and requirements of source code that makes up the project. The approach suggests that using this information could yield insight into the quality of the software. The approach also proposes an equation for quality based on the information stored in the Software Configuration Management system. In addition to proposing an approach to measuring quality, Chapter 4 presents a reference implementation of the approach, using a software package called Subversion as the Software Configuration Management System.

Chapter 5 presents and critically assesses a case study where the approach proposed in Chapter 4 is used in a software engineering project.

Chapter 6 concludes the work presented, critically assesses the presented approach and proposes future work.

Chapter 2

Background and concepts

This chapter sets the context of the dissertation and describes the concepts relevant to the topics under discussion. The core concept of quality is discussed, and standards and process models addressing quality are reviewed. Software engineering and software engineering process models are discussed. The concepts of software requirements, testing and defects are discussed. Metrics, and software metrics in particular, are researched with a view to understanding how metrics might provide insight into quality. Finally, software configuration management is discussed so that an understanding is built of how information providing insight into quality can be stored along with software configuration items in a software configuration management system.

The discussions of this chapter aim to provide the basis and understanding for the approach described in Chapter 4, where these concepts are used to propose an approach to measuring software quality based on information in a Software Configuration Management system.

2.1 Software Engineering

The Institute for Electrical and Electronic Engineers (IEEE) Standard Glossary of Software Engineering Terminology (IEEE Computer Society, 1990) defines *Software Engineering* as

> "(1) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software;
> (2) the study of approaches as in (1)."

Software Engineering is the set of tasks and processes that are applied to software in order to build and operate it. Software Engineering is more than just Software Development. Development is usually one of the activities required in a Software Engineering project, but Software Engineering encompasses all of the activities required to conceptualise, design, develop, test and properly run software.

2.2 Quality

As discussed by Garvin (1984), determining the quality of a product can be approached in a number of ways. The *transcendent approach* states that quality cannot be defined precisely, but is rather experiential, and one can learn to recognise quality by experience. The *product-based approach* holds that quality is measurable and related to the degree to which the product possesses some attribute. The *user-based approach* holds that quality is subjective, and is determined by the user of the product based on their perspective. The *manufacturing-based approach* equates quality to conformance to requirements: if a product is built as specified, it exhibits quality. The *value-based approach* holds that a quality product is a product that provides "performance at an acceptable price or conformance at an acceptable cost".

Different approaches to measuring quality are adopted in different situations. For example, the ISO 9000 (International Organization for Standardization, 2005) family of standards uses the *manufacturing-based* and *value-based* approaches.

It can be said that quality is a relative concept: the quality of one object may not be as high as the quality of another of the same type. Quality is also a subjective concept: what one person perceives as high quality may not be perceived as high quality by the next person.

Assuming that quality is both relative and subjective, there is an inherent difficulty in objectively quantifying the quality that an object possesses and proving objectively that something exhibits high quality.

The attempt to ensure quality, then, can be seen as a two-fold endeavour. Firstly, the quality of the product or service needs to be established as compared to other similar products or services. This is so that people will purchase and use the product instead of other similar products. Secondly, the quality of the product or service needs to be perceived as high by the largest possible proportion of people evaluating or using the product or service. This is so that people will purchase and use the product.

2.3 Industry Standards addressing quality

This section investigates three industry standards regarding quality. The standards discussed here were chosen because they represent a form of "suggested practice" based on experience. The ISO standards were chosen because they represent a distillation of experience in the field, as ISO develops standards that meet market needs, are based on global expert opinion, take input from various stakeholders and perspectives, and are based on consensus of these stakeholders (International Organization for Standardization, 2013). CMMI was chosen because it is the result of an academic collaboration with industry, and represents a similar distillation of experience and research. Each standard has its supporters and detractors, and some discussion is presented in this section regarding the perceived strengths and weaknesses of these standards.

- The ISO9000[1] range of standards focus on the Quality Management System of an organisation, and these standards are hence important to quality.
- The Capability Maturity Model Integration (CMMI) process model addresses quality by providing a process model framework which enables organisations to implement more effective processes which, it is claimed, leads to deliverables of consistent and increasing quality.
- The ISO25000 family of standards address software product quality evaluation and requirements specification.

2.3.1 The ISO9000 family of standards

The International Organization for Standardization (ISO) produced a set of standards known as the ISO 9000 family of standards. This family of standards consists of the following individual standards:

- ISO 9000:2005 Quality Management Systems — Fundamentals and vocabulary (International Organization for Standardization, 2005);
- ISO 9001:2000 Quality Management Systems — Requirements (International Organization for Standardization, 2000a); and

[1]The South African National Standards (SANS) version of the International Organisation for Standardization (ISO) specification has been used here, but references to ISO9000 and SANS900 documents are interchangeable.

- ISO 9004:2000 Quality Management Systems — Guidelines for performance improvements (International Organization for Standardization, 2000b).

The ISO9000 family of standards describes a process as "Any activity, or set of activities, that uses resources to transform inputs to outputs."

This standard states that the implementation and maintenance of a system that is designed to continually improve performance and address the needs of all stakeholders can add to the success of an organisation. The standard suggests that organisations that produce a product or provide a service use a *process approach*. This means that the organisation should view its activities as a set of processes and the set of interrelations and interactions between these processes. It is suggested that identifying and understanding these processes and their interactions will greatly enhance the organisation's capability to meet customer requirements consistently.

Examples of typical processes in software engineering projects are gathering requirements, developing software and testing. These processes have their own tasks, and possibly sub-processes, and they depend on, and interrelate with, each other and other processes in the organisation and vice-versa.

A Quality Management System is a system of processes in the organisation that is implemented and adhered to in the production of products or provision of services. Implementing a Quality Management System encourages an organisation to gather customer requirements, identify and define the processes that are required to produce a product or provide a service that will be accepted by the customer and keep these processes under control (in terms of managing change to these processes). The Quality Management System provides a framework for the continual improvement of these processes. This means that customer satisfaction should be enhanced and the confidence of the customer and organisation itself in the organisation's ability to consistently fulfil customer requirements should increase.

ISO9000 suggests that organisations follow eight Quality Management Principles.

1. Customer focus — customer requirements should be met and customer satisfaction should be strived for.
2. Leadership — the management of the organisation should ensure that there are no barriers to employees working towards the organisation's objectives.
3. Involvement of people — the skills of employees and management should be used to achieve the organisation's objectives.
4. Process approach — processes and their interactions should be identified and managed.

5. System approach to management — the organisation should see the interrelating processes in the organisation as a system so that objectives are more effectively and efficiently attained.

6. Continual improvement — as customer requirements and the market change, so should the performance of the organisation (and hence the processes used in the organisation).

7. Factual approach to decision making — the organisation should analyse data regarding the performance of the organisation (and processes used in the organisation) in order to make better decisions.

8. Mutually beneficial supplier relationships — because an organisation and its suppliers are dependent on each other, both parties benefit from a good relationship.

ISO9000 defines *quality* as

> "[the] degree to which a set of inherent *characteristics* fulfils *requirements*."

Further, *characteristics* are defined as

> "distinguishing features."

A *requirement* is defined as

> "a need or expectation that is stated, generally implied or obligatory."

So, to expand the definition, then, *quality* is:

> "the degree to which a set of inherent distinguishing features fulfils a need or expectation that is stated, generally implied or obligatory."

The definition given in this family of standards links the quality of a product or service to a set of requirements and characteristics of the product. The definition implies that the "better" a product or services matches the requirements specified when the product or service was being designed or conceptualised, the higher quality it exhibits. This view is adopted by the approach proposed in Chapter 4. This emphasizes the need for the organisation to understand and meet customer requirements. Also, because of the process approach that should be adopted, it is beneficial to consider the processes used to meet customer requirements in terms of the value that they add

to the organisation. Thus, there is a need to constantly monitor and measure process effectiveness and performance and improve the processes based on the outcome of the monitoring and measurement.

The ISO9001 standard gives a set of requirements for a Quality Management System. It states that there is a need to show that the product or service being produced or provided by an organisation *consistently* meets the requirements of not only the customer, but also any specific regulatory requirements that may exist for the product or service. In addition to this, the organisation should aim to enhance customer satisfaction through the continual improvement of the Quality Management System. The core tenets of the Quality Management System are itemised below.

- Identify, monitor, measure and ensure continual improvement of the processes used to produce the product or provide the service, and the interactions between these processes.
- Document the processes and ensure that this documentation is managed and well controlled. There is a need for a Quality Manual, which defines the scope of the Quality Management System and documents the processes and interactions thereof.
- Ensure that management is committed to (and accountable for) the implementation of the Quality Management System. Management should plan the Quality Management System and produce the Quality Policy, which provides a framework for the establishment and review of the organisation's quality objectives (i.e. things that are sought or aimed for relating to quality). Management should ensure that these documents (and their commitment to them) are clearly communicated to the organisation and that the responsibilities of all employees regarding quality and the employees' achievement of quality objectives are known.
- Ensure that employees who are involved with activities affecting product or service quality are appropriately trained and aware of their responsibilities.
- Plan and develop processes that will be required to realise the product or service. Identify the quality objectives, the processes, documentation and resources required and the activities (verification and validation, for example) that are required to demonstrate product acceptance (or put another way, fulfilment of customer requirements).
- Determine the processes required to elicit product or service requirements. Engage the customer and ensure that the requirements of the customer are well

understood and reviewed before committing to produce the product or provide the service.

- Ensure that the product or service is of consistent (or increasing) quality by measuring, analysing and improving the Quality Management System's performance.

The ISO9000 family of standards promotes a process approach to managing the quality of outputs of an organisation. This implies that if an organisation implements a system of processes (related to producing products or providing services) and ensures that there is a commitment to the implementation, maintenance and continual improvement of these processes, the outputs of the organisation will be of consistent (or improving) quality. This, in combination with a focus on the requirements and satisfaction of the customer, means that following this set of standards (and being certified against them) will enable an organisation to produce quality outputs that consistently meet customer requirements and continually aim to enhance the satisfaction of the customer.

As an example of the requirement for the existence of, and adherence to, process, the approaches to measuring software quality described in Chapter 3 and Chapter 4 all rely on process, and adherence to process, to measure and collect metric data to infer insights into quality.

2.3.2 CMMI

In any organisation involved in software production, there are three main areas of concern: People, Procedures and Tools (Phillips, 2006). The people need to be skilled (and maintain and develop their skills through training); the procedures define relationships between tasks that need to be performed; and the tools are used by the people to implement the procedures. However, these three areas are used most effectively when they are driven and directed by one or more process(es). The processes govern the people, describe how to implement procedures and give guidance about which tools to use and how best to use them.

The Software Engineering Institute (SEI) at the Carnegie Mellon University has developed a process improvement maturity model, Capability Maturity Model Integration (CMMI). CMMI is based upon The Process Management Premise (Chrissis et al., 2003):

> "The quality of a system is highly influenced by the quality of the process used to acquire, develop and maintain it."

Initially, the SEI developed a set of Capability Maturity Models (CMMs). These CMMs focused on process improvement, containing essential elements of effective processes for one or more disciplines. The purpose of the CMMs was to aid an organisation (specifically in their software engineering efforts) in the evolution from ad-hoc, immature processes to disciplined and mature processes. The result of this evolution was intended to be improved process quality and effectiveness.

A result of this work by the SEI was a set of CMMs for different disciplines. These include CMMs for:

- Systems Engineering;
- Software Engineering;
- Software Acquisition;
- Workforce Management and Development; and
- Integrated Product and Process Development.

However, over time, and as more CMMs became available and organisations needed to implement more than one CMM, it became unwieldy for organisations to do so. Organisations wanted to implement a single model that addressed the different areas of their business rather than implement multiple models, one for each area of their business. In response to this need, CMMI was devised.

CMMI aimed to integrate the CMMs so as to fulfil the requirement organisations had to implement a single model covering multiple areas of their business. CMMI groups CMMs into so-called *constellations*. These constellations group CMMI components by areas of interest. Currently, there are 3 constellations:

- CMMI for Development, which addresses software development and maintenance processes;
- CMMI for Services, which addresses processes used by organisations providing services internally or externally; and
- CMMI for Acquisition, which addresses processes used by organisations that acquire services or products internally or externally.

The CMMI for Development constellation (which is relevant to this dissertation) investigates process models covering project and process management, systems, software and hardware engineering and other supporting processes that are used in the

development and maintenance of software products and the provision of software services.

CMMI also aims to institutionalise the processes that are developed as part of the process improvement effort which assists organisations to improve their processes from ad-hoc to mature. When a process has been institutionalised, it becomes ingrained in the way that work is performed in the organisation and (perhaps more importantly) there is a commitment from senior management to the consistent performance of the process, the improvement of the process and the continual re-alignment of the process to business requirements.

When an organisation implements the guidelines set forth in CMMI, they are making a commitment to implement and improve processes in their organisation. This commitment implies that there is an intention to achieve some level of consistency in the outputs of activities. If the quality of the output was not high enough when the process was first implemented, the guaranteed re-alignment of the processes to business requirements means that the process will be adapted over time to ensure that the output is indeed of higher quality. Moreover, it will ensure that the quality of the output is of a consistent and increasing level.

The view of process that the quality of a system is influenced by the quality of the processes used to develop it is relied upon by the approaches discussed in Chapter 3 and Chapter 4.

2.3.3 The ISO25000 family of standards

The ISO25000 (International Organization for Standardization, 2007a) family of standards[2], consists of five divisions:

- Quality Management Division, which contains standards such as "ISO25000: Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE" (International Organization for Standardization, 2007a);
- Quality Model Division, which contains standards such as "ISO25010: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models" (International Organization for Standardization, 2011b);

[2]The SANS version of the ISO specification has been used here, but references to ISO25000 and SANS25000 documents are interchangeable.

- Quality Measurement Division, which contains standards such as "ISO25020: Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Measurement reference model and guide" (International Organization for Standardization, 2008b);
- Quality Requirements Division, which contains standards such as "ISO25030: Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Quality requirements" (International Organization for Standardization, 2008d);
- Quality Evaluation Division, which contains standards such as "ISO25040: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Evaluation process" (International Organization for Standardization, 2011a); and
- SQuaRE Extension Division, which contains standards such as "ISO25051: Software engineering - Software product quality requirements and evaluation (SQuaRE) - Requirements for quality of Commercial Off-The-Shelf (COTS) software product and instructions for testing" (International Organization for Standardization, 2007b).

The ISO25000 family was introduced as a replacement of two previous standards: ISO9126 (International Organization for Standardization, 2003) and ISO14598 (International Organization for Standardization, 1999), which dealt with quality models and product evaluation, respectively. Much of the terminology and concepts present in these two standards are still present in the ISO25000 family of standards. The ISO25000 family of standards is also collectively known as "Software product quality requirements and evaluation" (SQuaRE).

ISO25000 describes the lifecycle of software product quality. Internal software quality describes the quality of the product when it is under development. External software quality describes the quality of the software when it is in operation. Quality in use describes the quality of the software when it is being used. It also describes that the lifecycle of the implementation of software quality is similar to the software development process, in that there are requirements, implementation and validation phases. More details on software developement processes can be found in Section 2.4 and verification and validation are discussed in Section 2.5.3. This lifecycle model is depicted in Figure 2.1.

ISO25010 describes a framework for quality models that categorize product quality into characteristics, which may be sub-divided into subcharacteristics and possibly further into sub-subcharacteristics. The properties of a product that relate to quality

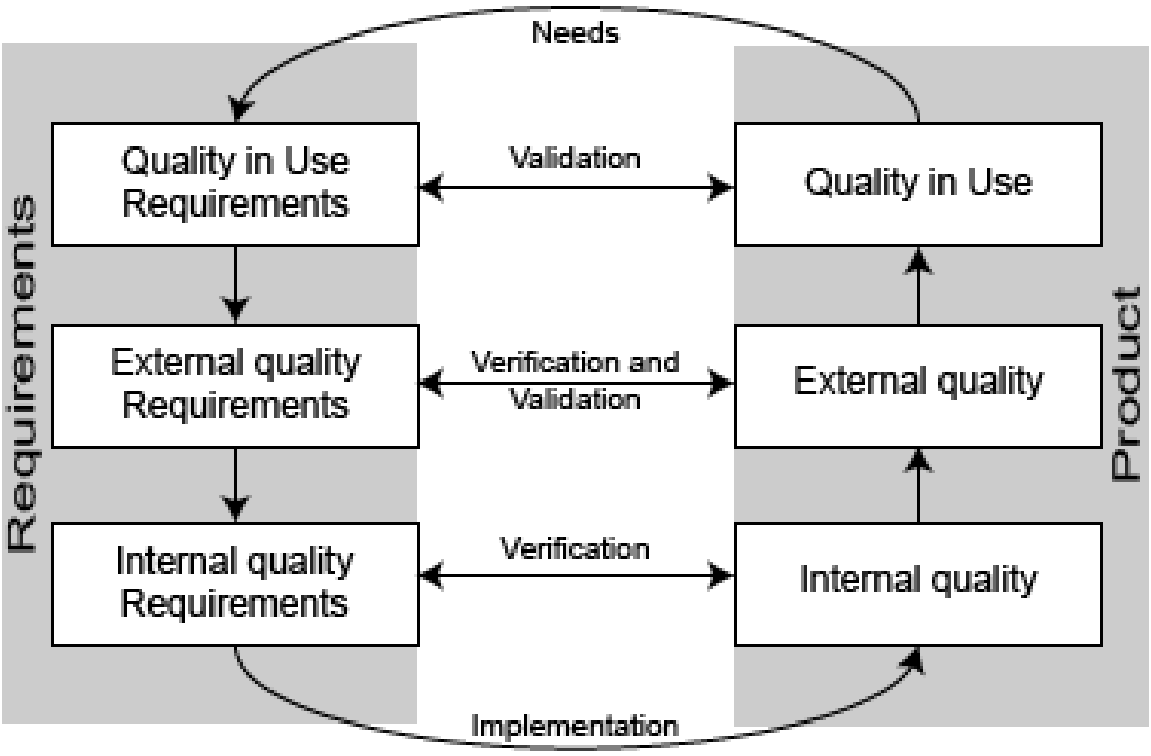

Figure 2.1: ISO25000 Software Product Quality Lifecycle Model

are called quality properties and are measured by quality measures. Characteristics or subcharacteristics may be broken down into these quality properties to aid measurement. ISO25010 defines two instances of the model framework: the Quality in use model and the Product quality model (with a third quality model, the Data quality model being the subject of a separate standard, ISO25012 (International Organization for Standardization, 2009)). Product Quality comprises internal quality and external quality (as measured by internal and external quality measures, respectively).

As suggested in ISO9126 (upon which the ISO25000 family is based), and in ISO25010, the quality model suggests that in any phase of a development project's lifecycle, the quality of the process (i.e. the process of the lifecycle) influences the product quality, which influences the quality in use. This means that the quality of the process used to develop the software is integral to the quality of the product as experienced by end users. Because the quality of the product can be measured using internal quality measures, external quality measures and quality in use measures, it is possible (and advisable) to allow feedback from users to developers to the process.

The relationship between process quality, internal quality, external quality and quality in use is summarised in Figure 2.2. This figure is adapted from the ISO9126 and

ISO25010 standards.

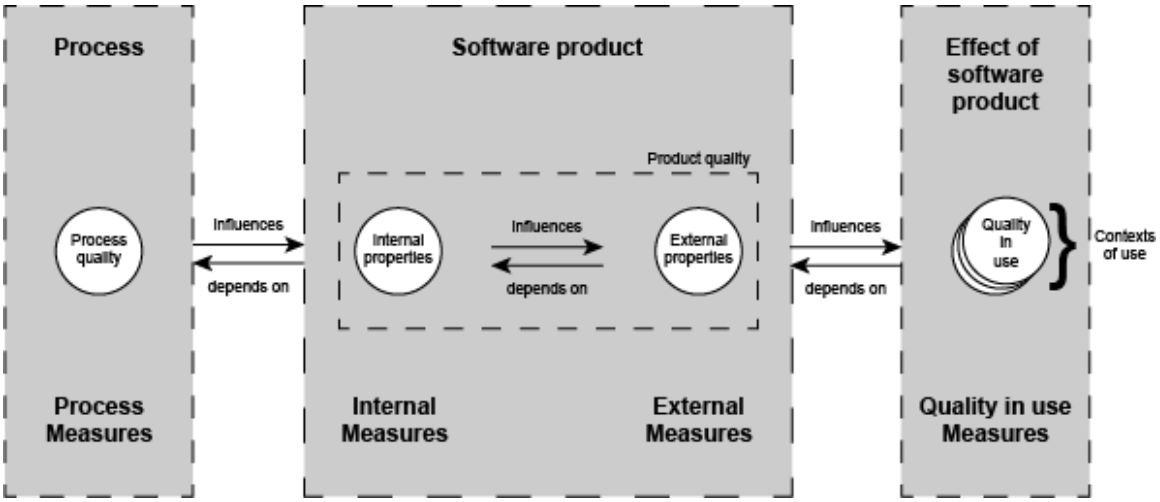

Figure 2.2: ISO25010 Quality Model

The product quality model presented in ISO25010 focuses on eight characteristics of software quality, which are further specified as sub-characteristics. The characteristics can be seen as somewhat independent — for example, it is clear that high reliability could be achieved without high usability. However, the sub-characteristics all relate to the characteristics which they are intended to measure. The list below describes these eight characteristics and their sub-characteristics.

Unless otherwise stated, it is assumed that the product is used under specified conditions. The formal definition of the metrics are given in ISO25010, and the definitions may be paraphrased here for ease of understanding. Where product is mentioned, it can be assumed that the statement applies equally to a system.

- Functional suitability: how well the functionality provided by the product meets stated and implied needs.
  - Functional completeness: how well the functionality provides for all of the specified tasks and user objectives.
  - Functional correctness: how well the product provides correct and precise results.
  - Functional appropriateness: how well the functionality helps achieve specified tasks and user objectives, or how suitable the product is for the tasks and user objectives.

- Performance efficiency: how well the product performs given the amount of resources it consumes.
  - Time behaviour: the requirements for processing times when performing functions are met.
  - Resource utilization: the requirements for the amounts and types of resources the product or system uses are met.
  - Capacity: The limits of the product or system meets requirements.
- Compatibility: how well the product is able to communicate with other products or systems as well as perform its own functions using the same hardware or software.
  - Co-existence: how well the product can perform its own functions while not negatively impacting other products.
  - Interoperability: how well the product can exchange information with, and use information from, other products or systems.
- Usability: how effective, efficient and satisfying a product is to users in performing their tasks and objectives.
  - Appropriateness recognizability: how easy or hard it is for users to assess the functional appropriateness of a product.
  - Learnability: how well the product assists the user in learning how to use it effectively, efficiently and with freedom from risk.
  - Operability: how well the product assists the user in operating and controlling it.
  - User error protection: how well the product protects users against making errors.
  - User interface aesthetics: how pleasing and satisfying the user interface is for the user.
  - Accessibility: how well the product lends itself to being used effectively by people with a wide range of characteristics and capabilities.
- Reliability: how well the product maintains an acceptable level of performance and users can rely on the software to perform the tasks required of it.
  - Maturity: how well the product meets needs for reliability under normal operation.

  - Availability: the product is operational and accessible when required for use.
  - Fault Tolerance: the ability of the product to maintain a specified level of performance in the face of faults in the software.
  - Recoverability: the ability of the product to get back to a specified level of performance and ensure that data is not lost when a failure occurs.

- Security: degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization
  - Confidentiality: how well the product ensures that data are accessible only to those authorized to have access.
  - Integrity: how well the product prevents unauthorized access to, or modification of, computer programs or data.
  - Non-repudiation: how well the product can assist in proving that certain events have taken place so that the event or actions cannot be repudiated later.
  - Accountability: how well the product associates an entity's actions with that entity.
  - Authenticity: how well the product can prove that the identity of a subject or resource is as claimed.

- Maintainability: the product is adaptable to changes in the environment, requirements and other changes.
  - Modularity: the product is composed of discrete components such that a change to one component has minimal impact on other components.
  - Reusability: parts of the product can be used in more than one product, or in building other products.
  - Analysability: the ease with which causes of failures in the product can be identified and the ease with which the impact of changes on the product can be understood.
  - Modifiability: the ability to change the product with relative ease, without decreasing quality or introducing new defects.
  - Testability: the ease with which tests can be created for the product and the ease of determining if a test was successful or not.

- Portability: how easily the product can be transferred from one environment to another.
  - Adaptability: the ease with which the product can be adapted to different environments.
  - Installability: the ease with which the product can be installed or removed.
  - Replaceability: the ability of the product to be used in place of another product for the same purpose in the same environment.

For the quality in use model, the characteristics are listed below.

- Effectiveness: accuracy and completeness with which users achieve specified goals.
- Efficiency: the resources that the product uses in achieving the user's goals accurately and completeley.
- Satisfaction: how satisfied a user feels when using the product.
  - Usefulness: how satisfied users are with the results and consequences of the use of the product.
  - Trust: how confident the users or stakeholders are that the product will behave as intended.
  - Pleasure: the degree to which a user obtains pleasure from fulfilling their personal needs to provoke pleasant memories.
  - Comfort: how satisfied the user is with physical comfort.
- Freedom from risk: how well the product mitigates the potential risk to economic status, human life, health, or the environment.
  - Economic risk mitigation: how well the product mitigates the potential risk to financial status, efficient operation, commercial property, reputation or other resources.
  - Health and safety risk mitigation: how well the product mitigates the potential risk to people.
  - Environmental risk mitigation: how well product mitigates the potential risk to property or the environment.
- Context coverage: how well the quality in use of the product translates when it is used in contexts of use that are both specified and those that are not.

- Context completeness: the quality of use of the product in all the specified contexts of use.
- Flexibility: the quality of use of the product in contexts of use other than those specified.

These characteristics and sub-characteristics provide a checklist of areas that software producers can focus on (with the appropriate metrics and actions on those metrics) to assist in attaining quality in their products: internally, externally and in use. This represents one approach to assessing the quality of a software product. The ISO25010 standard is not prescriptive about how the characteristics should be measured, leaving the determination of the value of a certain characteristic or sub-characteristic up to the organisation. The ISO25020 standard, does provide some guidance on the measurement, though. In addition, it may not be practical to apply all of these characteristics to all software projects (given time and resource constraints) or apply the quality in use characteristics to all possible use cases for the software product. These characteristics and sub-characteristics make up a suggested area of focus when determining quality, and organisations should choose the most applicable given their context.

2.3.4 Discussion of standards

It is important to bear in mind that industry standards are not the definitive instruction on a matter - they are merely guidelines based on experience. Thus, it is natural to expect that there are supporters and detractors for these standards.

Stelzer et al. (1997) discuss various views of ISO 9000 and how effective it is at improving product quality. The article addresses some of the criticisms that the ISO 9000 family has faced. Such criticisms include that implementing ISO 9000 does not necessarily lead to better software products, that implementing ISO 9000 can be a burdensome and expensive undertaking, and the return on investment is low as a result. According to ISO 9000, there are two motivations for implementing it: the customers or stakeholders demand it, or the management of the company demands it. In a survey of companies that have implemented ISO 9000, Stelzer et al. (1996) found that the majority of companies surveyed chose to implement ISO 9000 because there would be some competitive advantage to implementing ISO 9000 or that it was a prerequisite to participating in the market. Furthermore, they found that even though these companies did not implement all of the guidelines of ISO 9000, they found the implementation of ISO 9000 valuable to their companies.

It is not empirically clear that implementing ISO 9000 will improve the quality of products that an organisation produces. There is argument, though, that the process

of implementing ISO 9000 is beneficial to the organisation, and that having an ISO 9000 certification is beneficial to the organisation's competitiveness in the market.

In two similar surveys, Staples et al. (2007) and Khurshid et al. (2009) investigate why organisations may decide against implementing CMMI. The main reasons that these surveys uncovered are that even though the potential benefits of Software Process Improvement (Gibson, 1998) might be attained, such as standardised processes, higher product quality, better reaction to defects and reduced time to market, organisations find CMMI too costly, they perceive their business as too small, or they don't perceive that the benefit of implementing is worth the cost and would rather focus on other activities that have higher perceived priorities. In an analysis of an early version of CMMI, Pierce (2000) found that CMMI was focussed on large projects, and had a large number of areas for implementors to address, and that this would cause some barrier to entry for smaller organisations.

While an organisation does stand to gain from the benefits of Software Process Improvement, there is evidence suggesting that small organisations do not see these benefits by implementing CMMI - in fact, CMMI itself may present a barrier to their attaining these gains.

The ISO25000 family of standards is a combination of ISO9126 and ISO14598. ISO14598 had been criticised because it was so closely related to ISO9126, and a unified standard would cause less confusion and promote adoption, according to Azume (2001). ISO9126 has been criticised by Al-Qutaish (2009) for many reasons, one of which being an ambiguous structuring of characteristics and sub-characteristics. Vlas (2011) proposes amending the ISO25000 standards by adding explicit representation of requirements specifications to the measurement reference model and specifically mentioning the significance of complementary services and components by adding them to the product quality model. The claim is that adding these two items to the standards would allow for a more accurate representation of people's real perception of quality.

It is clear that there were areas of improvement for ISO9126 and ISO14598, and the ISO25000 family was created to address those. Since the ISO25000 family is fairly new, there isn't much evidence about its efficacy in increasing product quality, but it is presented as a positive step forward from ISO9126 and ISO14598.

The standards and guidelines that have been discussed in this section have been created as a result of research and work in the area of quality in general, and quality in software engineering in specific. A theme that emerges from the discussion of these standards is an approach to quality by emphasising the value of *process*.

The more generic ISO 9000 family of standards promotes the use and maintenance of a Quality Management System, suggesting a focus on (among other things) a process approach, customer focus and continuous improvement.

CMMI focuses on the processes involved with developing software and improving these processes over time.

The ISO25000 family of standards proposes a quality model and a set of metrics that can be used to attempt to objectively assess the quality of a software product and suggests that the quality of the process used to develop software influences the quality of the product as perceived by the users of the product.

The process view and quality models described in this chapter give context to the discussions of quality metric models discussed in Chapter 3, and the approach proposed in Chapter 4, which rely upon the understanding of these concepts to provide insight into quality using metrics.

The following section will investigate software engineering and various process models that have been created to approach the tasks of a software engineering project.

2.4 The Software Engineering Process

Following on from the emphasis on the value of *process* discovered in the previous section, this section describes how this has been made concrete in the day-to-day activities of software engineers, and how process forms an integral part of producing software. By discussing these process models, it should become clear that extending these models, or adding to the tasks of these models should be possible and that gathering and analysing metrics relating to software quality could become just another step in the process.

The IEEE definition of Software Engineering (IEEE Computer Society, 1990) implies that there are three areas of concern in Software Engineering:

- development;
- operation; and
- maintenance.

This dissertation deals primarily with the development activities of a software project. The development activities are described by a process. The software development process (also referred to as the software development life cycle) describes a set of tasks and activities that must be followed while developing or constructing software. The "Software Implementation Processes" section of the ISO 12207 (International Organization for Standardization, 2008a) standard[3] defines the following umbrella processes as part of the implementation or construction of a software product:

- Software Requirements Analysis;
- Software Architectural Design;
- Software Detailed Design;
- Software Construction;
- Software Integration; and
- Software Quality Testing.

Each of these processes can be viewed as a set of tasks or activities (i.e. sub-processes) that are performed as part of creating a software product. Each project and organisation may choose to use a different set of processes or process model. The sequence of the processes within the development life cycle are different per model, but they should all address the processes described by ISO12207.

According to Booch (1995), a software development process has four roles:

- provide guidance as to the order of a team's activities;
- specify which artifacts should be developed and when they should be developed;
- direct the tasks of individual developers and the team as a whole; and
- offer criteria for monitoring and measuring the project's products and activities.

[3]The South African National Standards (SANS) version of the ISO specification has been used here, but references to ISO12207 and SANS12207 documents are interchangeable.

This section discusses a broad categorisation of software process models based on those described by Pressman (2005). For each type of process model, an example of that type of process model will be discussed. The processes and activities discussed here provide a basis for the discussions of Chapter 3 and Chapter 4.

2.4.1 Linear Process models

Linear process models are those that view the activities in the software development life cycle as a sequential set of steps. An example of a linear process model is the Waterfall Model.

When Winston W. Royce described a way of managing large software development projects in his paper titled "Managing the development of large software systems" (Royce, 1970), he was (whether this is what he intended or not) actually laying the foundation for one of the first formal software engineering processes, the Waterfall Model. The Waterfall Model is characterised by a sequential set of stages in the lifecycle of a software development project, and is thus categorised as a linear process model. Each stage of the software development life cycle feeds directly into the next, with no iteration between steps[4]. Royce's paper suggests that the basic steps of the software development process for small, internal products (i.e. those for internal consumption) are analysis and coding, but in larger software projects (and those destined for external customers), there is a need for more steps. These steps are System Requirements, Software Requirements, Analysis, Program Design, Coding, Testing and Operation. The Waterfall Model assumes the following (Leffingwell, 2007):

- there is always a well-defined set of requirements that are "set in stone" at the beginning of the project;
- if changes are made to requirements, they will be small enough to integrate into existing work without affecting the delivery of the project;
- integration of software deliverables is predictable; and
- effort in Research and Development projects can be accurately predicted.

The quality of the resulting product, then, is assumed based on these assumptions. If the requirements are well-defined and integration is predictable and thus more easily

[4]It should be noted that Royce presented this model as one that is inherently flawed and suggested iteration between the phases of the development process.

managed, all that needs to be done is for the developers to follow the specifications set forth in the requirements (and occasionally make patches to code in the event of changing requirements) for the product to be exactly what the customer wants and hence be of high quality.

Royce addresses these assumptions in his paper. Some of the problems uncovered in the Testing phase of a project are the kind of issues that require a change to the design of the product. This means that the process needs to return to the coding phase, which could cause a large delay to delivery. This contradicts the assumption that changes to requirements will not delay the project. Also, Royce highlights the fact that certain aspects of the product's functioning can actually only be realised in the practical Testing phase (as opposed to the theoretical Analysis phase). Things like storage requirements, processor usage and so on are really only fully observed at testing time. Royce proposes five steps to overcome the downfalls of the waterfall model.

1. Design the program before the Analysis phase. The program that is being developed should take into account things like storage and processor usage up front so that these operational issues (i.e. those issues that are not realised until the software is actually executed) are not only uncovered in the Testing phase. This will ensure that the system that is being developed will be documented and understood and aspects of operations and design will be documented early in the development cycle.

2. Document the design. Effective design documentation will mean that the design is contracted and it is understood what the requirements of the system being developed are. This helps in the Testing phase, where testing becomes easier because requirements are clear and it helps in the Operations phase, because it will form a basis for operating procedures, redesign and updates.

3. Do it twice. Delivering a pilot or prototype version of the system will uncover problems before the final version is delivered, giving a chance to rectify these problems before release.

4. Plan, control and monitor testing. The testing phase of a project is often the biggest phase in terms of resource requirements (both hardware and human). This makes the Testing phase the biggest risk to the project. It is suggested that independent testers (people not involved in the design or analysis phases) test the code and that every logic path in the code is tested.

5. Involve the Customer. If the customer is involved and committed from the beginning, the difference between what the designer believed the customer wanted

and what the customer actually wanted will be smaller. Thus, there is a higher chance that the customer's requirements will be met.

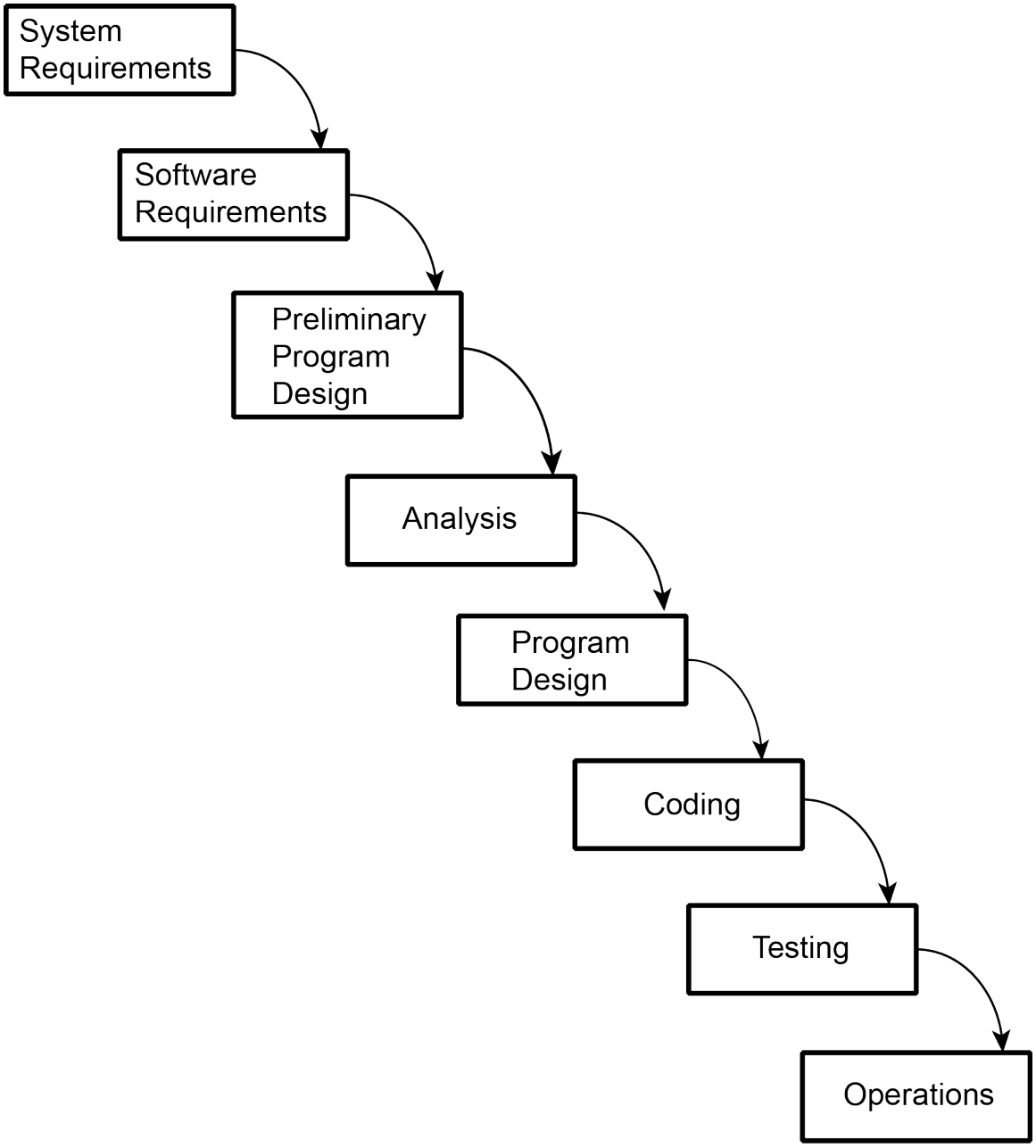

Figure 2.3: The Waterfall Model

Figure 2.3 depicts the waterfall model described in Royce (1970). Royce's suggested

alterations to the Waterfall Model go a lot further than the original to ensuring that the customer is satisfied. By ensuring that functional issues are addressed and by focussing on finding the problems in the developed system (and indeed the documented design) early in the process and by involving the customer, the foundation is set for development that meets customer requirements and leaves a customer satisfied with the outcome. This implies that, while the initial Waterfall Model innocently assumed that quality would be attained as a result of the (false) assumptions of the Waterfall Model, Royce's alterations to the model actually address quality more definitely.

The ideas of interim products, or prototypes, and regular testing, are used in the approach discussed in Chapter 4.

2.4.2 Incremental Process Models

Incremental process models are similar to linear process models, in that the steps in the development life cycle are performed sequentially from the first to the last. The difference between linear and incremental process models is that incremental process models execute the development process more than once. The sequence of steps that make up the life cycle may be executed more than once. The successive executions of the life cycle provide more and more of the functionality required of the product. The first increment delivers a "core" of the product, with many features unaddressed. After this increment, more features are added to the product in successive iterations. Each increment may be presented to the customer, and feedback from the customer, in terms of requirements and modifications to the design, are catered for in the next increment. Each increment is supposed to produce a working product, with certain functionality enabled.

An example of an incremental process model is the Rapid Application Development Model (RAD) (Martin, 1991). The RAD Model focusses on short duration product increments with some of the requirements being specified up front, and the design and coding steps being performed in iterations. Each iteration's result is tested and evaluated against requirements. If changes to the design are required, they are performed and worked into the requirements for the next iteration. Each iteration of the design and coding phase yields an interim product that is more close to being complete than the previous iteration. Once the product is verified to meet requirements, it is deployed. With a RAD Model, the focus is on prototyping in order to incrementally refine the design and rapidly build and verify required functionality. Figure 2.4 depicts the RAD Model.

The idea of interim products is used in the approach discussed in Chapter 4.

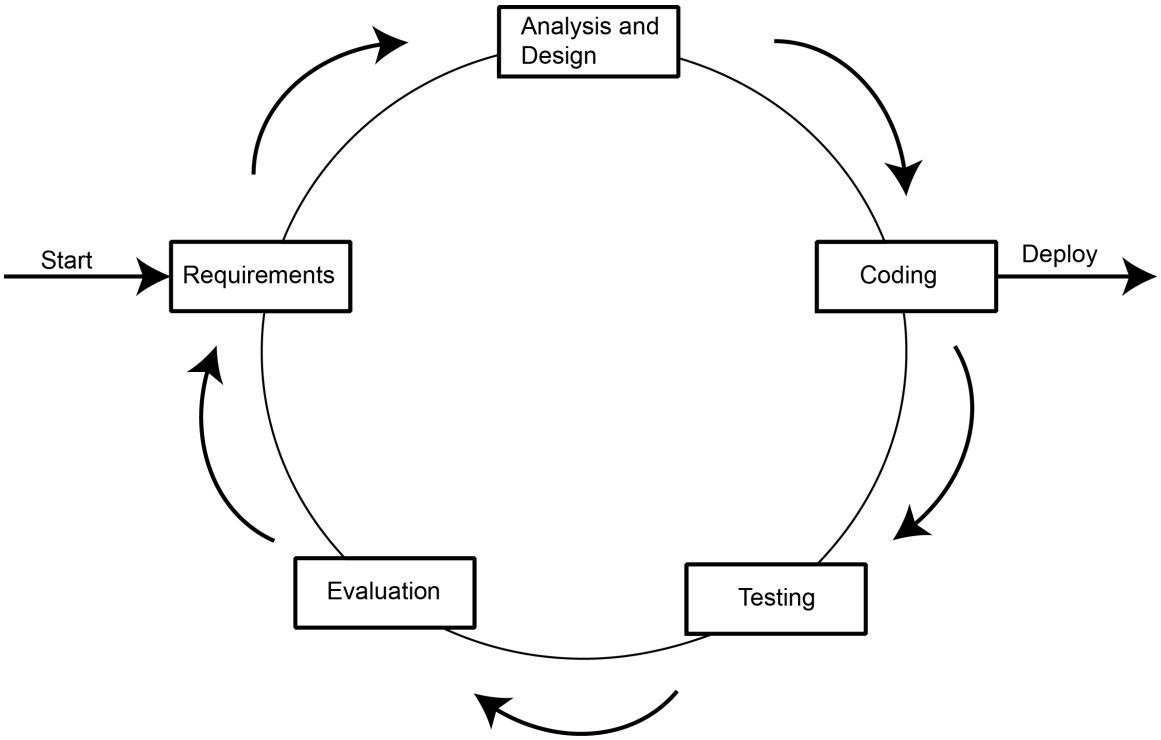

Figure 2.4: The Rapid Application Development Model

2.4.3 Evolutionary Process Models

Evolutionary process models are iterative in nature. These process models allow for changing requirements and cater for products that evolve over time. Evolutionary process models differ from Incremental process models in that incremental process models focus on releasing working products for customer consumption after each iteration, where evolutionary process models generally evolve a full product over time.

An example of an evolutionary process model is the Spiral Model.

The Spiral Model of software development and enhancement (as named by Boehm (1986)) is a risk-driven approach (as opposed to a document-driven or code-driven approach) to the development of software. The Spiral Model views the process of software development as an iterative set of activities, or a set of cycles. Each cycle is concerned with the elaboration of a specific portion of the deliverable product. The objectives of the chosen portion are identified, along with alternatives for implementing the portion and the constraints associated with each of the alternatives. Each of the alternatives is evaluated taking into account their objectives and constraints, which will give some idea of the risks associated with each of the alternatives. Here, the "risk" is the possibility of the alternative not meeting the desired objectives and the constraints causing requirements not to be met.

Once the relative risks are identified, steps are identified to resolve or mitigate the risks. Such steps depend on the level of the risk, but can include prototyping. Once the risks are identified and resolved, the next iteration of the product can be developed (including the portion of the product currently under elaboration) and verified, using the development process of detailed design, requirements specification, design validation, development, testing and implementation. After the iteration of the product is complete (i.e. is tested and implemented), the next cycle of the process is planned. This planning encompasses requirements, life-cycle, development and integration, and testing for the next portion of the product to be elaborated.

The cycle finishes with a review of the activities in the cycle by the stakeholders of the product (which should include the customer). The review will determine whether the output from the previous cycle met the requirements for the cycle, bearing in mind that the outputs may not have included a software deliverable at all, and could be limited to documentation or other interim deliverables. The review also determines the plan and resources required for the next cycle.

Due to the fact that this process is driven by risk, risk management techniques are used to determine the amount of time and effort that are expected to be required for planning, configuration management, quality assurance, and formal verification and testing. The level of risk determines the degree of completeness, formality and

granularity of the specifications for the product. The relative risk of over-or-under specifying the product assists in determining this. A disadvantage of the risk-driven approach is that there needs to be experienced personnel to apply these risk management activities.

The point of departure in the spiral model is the hypothesis that some operational mission could be improved by a software development effort. The process ends when this hypothesis is false. The hypothesis can be rendered false if the software is no longer required, the risk of implementing is too high or the product has been implemented successfully. Figure 2.5 depicts the Spiral Model.

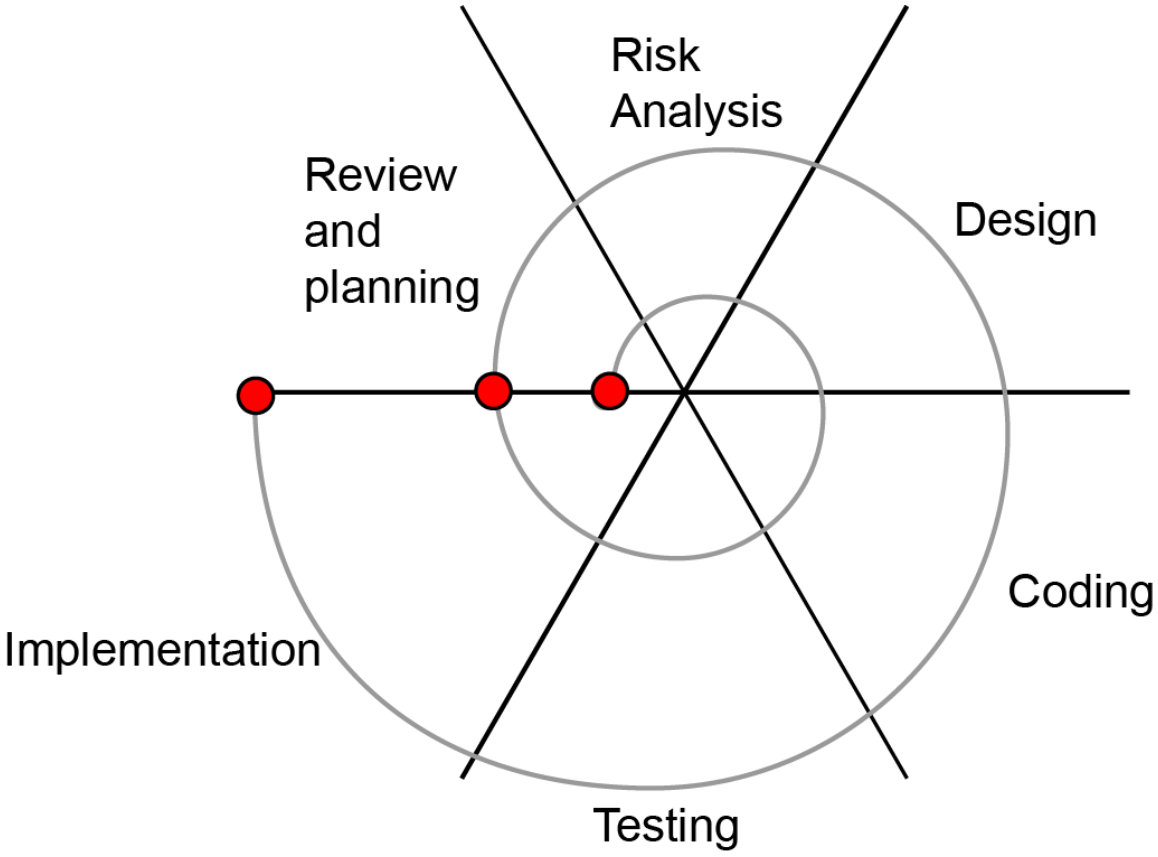

Figure 2.5: The Spiral Model

The Spiral Model of software development bases its activities on risk: the risk that implementing a certain alternative may cause delays to the project timelines or may adversely affect customer satisfaction. This focus on risk means that the output of the development effort is more likely to meet the requirements of the customer because alternatives not meeting requirements would have been identified and dealt with in

the cycles of the spiral model. Although the spiral model does not explicitly address quality, it does ensure that the major risks are dealt with, which in turn implies more customer satisfaction, which has the same effect as explicitly addressing quality output.

The ideas of interim products, or prototypes, and regular testing, are used in the approach discussed in Chapter 4.

2.4.4 Specialised Process Models

Specialised process models are those that apply a specific software engineering approach, where the process and approach is prescribed by the model. An example of a specialised process model is the Formal Methods Model, a variation of which is Cleanroom Software Engineering.

Cleanroom Software Engineering was conceived and formalised by researchers working at IBM. The origin of the Cleanroom method is a paper titled "Certifying the reliability of Software" (Currit et al., 1986). In this paper, Currit et al. discuss a method of certifying the reliability of software before the software is released to users. The approach uses executable product increments, statistical testing of the most common user functions of the product and a certified estimate of the Mean Time To Failure (MTTF) of the software. In a traditional development life cycle, design, testing, defect removal and implementation are performed in a phased manner. However, these phases alone cannot prove how reliable the end product is, nor can they say with certainty that after removing defects, none remain or none have been introduced as part of the defect removal effort. An assumption made by Currit, et al. is the fact that users of the software being produced will not be as concerned with the number of defects in the software as they will be with the reliability of the software. Users want to know how long the software will work without a problem before it fails and what the operational impacts of a failure are. Increasing the time to failure (the MTTF) for a product means that customer satisfaction could be increased because the product could be used for a longer period of time without failure.

The focus of activity in this approach is in the testing phase. It is suggested that the testing group is a group distinct from the developers of the product. After each executable increment is developed, the testing group will attempt to certify the reliability of the product. It is assumed that the output of the development group exhibits high quality, so the focus is not on quality but rather on certifying reliability. The test cases that the test group execute are based on a statistical sampling of the user operations that would be performed on the product. Testing is done in environments that closely match the intended release environment. Testing failures naturally represent a departure in the software from the structured specification that is required

to base development on. The results of the testing process are used to confirm or modify the development practices used to develop the product increments.

Building on this initial work on software reliability, Mills et al. (1987) worked on the assumption Currit et al. made that the development output exhibits high quality by proposing a more formal approach to the development effort. The result of this work was the Cleanroom Software Engineering approach. Cleanroom is a theory-based, team-oriented approach using statistical quality control to develop very high quality software. The approach combines formal methods, function-theoretic correctness verification and statistical usage testing for quality certification.

Cleanroom uses mathematical foundations in the design, specification and correctness verification of the product. The focus on a formal proof that the software works as required eliminates the need to debug developed outputs. The use of formal methods of object-based box-structure specification and design ensures that the specification and design of the product are well-defined. Function-theoretical methods are used to verify the correctness of software. Statistical usage testing is used to certify the quality of the product.

While the initial work on Cleanroom focussed on statistical testing and verifying the reliability of software, the assumption it made that the developed increments were of high quality was a problem. Further work on Cleanroom introduced the idea of using mathematical techniques not only to verify the reliability of the software but to ensure that the specification, design and developed product were of high quality. The idea of using tests that cover the most-often used functions of the product is used in the approach discussed in Chapter 4.

2.4.5 The Unified Process

The Unified Process is a use-case driven, architecture-centric, iterative and incremental software process (Jacobson et al., 1998).

The Unified Process places importance on the requirements of the customer and the role of the software architect. The Unified Process views software development as a set of phases. These phases are detailed here.

- Inception. The inception phase involves gathering customer requirements, providing a first-draft architecture of the software and planning the implementation of the software.
- Elaboration. The elaboration phase involves expanding on the requirements gathered in the inception phase and creates architectural views of the system.

- Construction. The construction phase involves actually building the software, and executing unit tests against the developed software.
- Transition. The transition phase involves user testing and defect reporting to the development team.
- Production. The production phase involves deploying and supporting the end software products and managing changes to the deployed software.

Each of the phases of the process may be iterated over. Each iteration produces an increment of the output, which expands on the previous iteration's increment. It may not be the case that all of the phases are executed in parallel, but there may be a "staggered concurrency" of phases, where each successive phase starts a short time after the previous phase has started.

An example of the Unified Process is the Rational Unified Process (RUP) (Kruchten, 2003).

The Rational Unified Process[5] is built on the principles of the Unified Process, and suggests adherence to "Software Best Practices" in order to avoid failure of projects due to common root causes. These best practices are described below.

- Develop software iteratively. This allows for user feedback and testing early in the life of the product and provides information and progress indicators to management and stakeholders early in the project.
- Manage requirements. This allows the project team to deliver what is really required.
- Use component-based architectures. This increases the resiliency of the product's architecture and aids reuse, modularity and configuration management.
- Visually model software. This provides for an unambiguous view of the architecture of the product.
- Continuously verify software quality. Using automated testing capabilities provides an objective view of the project's status from early in the development life cycle.
- Control changes to software. This makes the team development effort more effective.

[5]The Rational Unified Process is based on the Unified Process, and was developed by Rational Software, now owned by IBM

The Rational Unified Process builds on the six best practices and provides a process for software engineering that is "focused on ensuring the production of quality systems in a repeatable and predictable fashion." (Kruchten, 2003). Kruchten states that "A process describes who is doing what, how, and when" (Kruchten, 2003) and that the Rational Unified Process is represented by five primary elements, described below.

- Roles, which represent the "who". Roles define the behaviours and responsibilities of an individual or group of individuals in a team.
- Activities, which represent the "how". Activities are units of work that an individual in a role performs.
- Artifacts, which represent the "what". An artifact is a piece of information that is produced, modified or used by a process. Artifacts form input to and output from activities.
- Workflows, which represent the "when". A Workflow is a sequence of activities that produces a result of observable value.
- Disciplines, which are containers for the other elements. Disciplines represent a partitioning of all roles and activities into logical groupings by areas of concern or specialty.

The Rational Unified Process has nine core disciplines.

- Business modelling, which is concerned with understanding the business environment in which the product is to be deployed, understanding the business need for (and value of) the system, and creating a set of system requirements that are understood by all parties.
- Requirements, which aims to provide understanding and agreement by all parties of the scope of the system. The output of this phase allows for the technical and financial planning of the project. User interfaces are designed in this discipline.
- Analysis and design, which will translate the requirements into a specification that will describe how the system will be implemented.
- Implementation, which is concerned with the actual development (or programming) of the code for the product, and unit testing of this code.
- Test, which focuses on evaluating or assessing product quality.

- Deployment, which is concerned with delivering the finished software product to the user.
- Project management, which aims to manage resources, planning and risk in software projects.
- Configuration and change management, which will attempt to ensure that the integrity of the project assets (code, documentation, etc.) is managed in the face of changes to these assets.
- Environment, which supports the development teams with processes and tools.

Each of these disciplines is exercised to some degree during one or more of the phases of a project following the Rational Unified Process. The phases are derived from those defined in the Unified Process. Each of the phases concludes with a milestone. The phases and their related milestones are enumerated below.

1. The inception phase concludes with the lifecycle objective milestone.
2. The elaboration phase concludes with the lifecycle architecture milestone.
3. The construction phase concludes with the initial operational capability milestone.
4. The transition phase concludes with the product release milestone.

An iteration of these phases is called a development cycle and results in a software generation. Successive iterations of the phases yield further generations of the software. Each iteration will place different emphasis on each of the phases. Similarly, the focus of activities will shift for each of the iterations. An example of this is that a larger proportion of the design of the product will be performed in the earlier generations of software.

Figure 2.6 depicts the Rational Unified Process Model. The figure indicates the phases of the project and the effort spend during each phase on each of the core disciplines (as represented by the shaded area).

The Rational Unified Process is an iterative process that values architecture and modelling in order to describe a software product. The process consists of phases and disciplines, which segment the activities of the software lifecycle.

The ideas of interim products, requirements management, continuously measuring software quality and controlling change to the software are used in the approach discussed in Chapter 4.

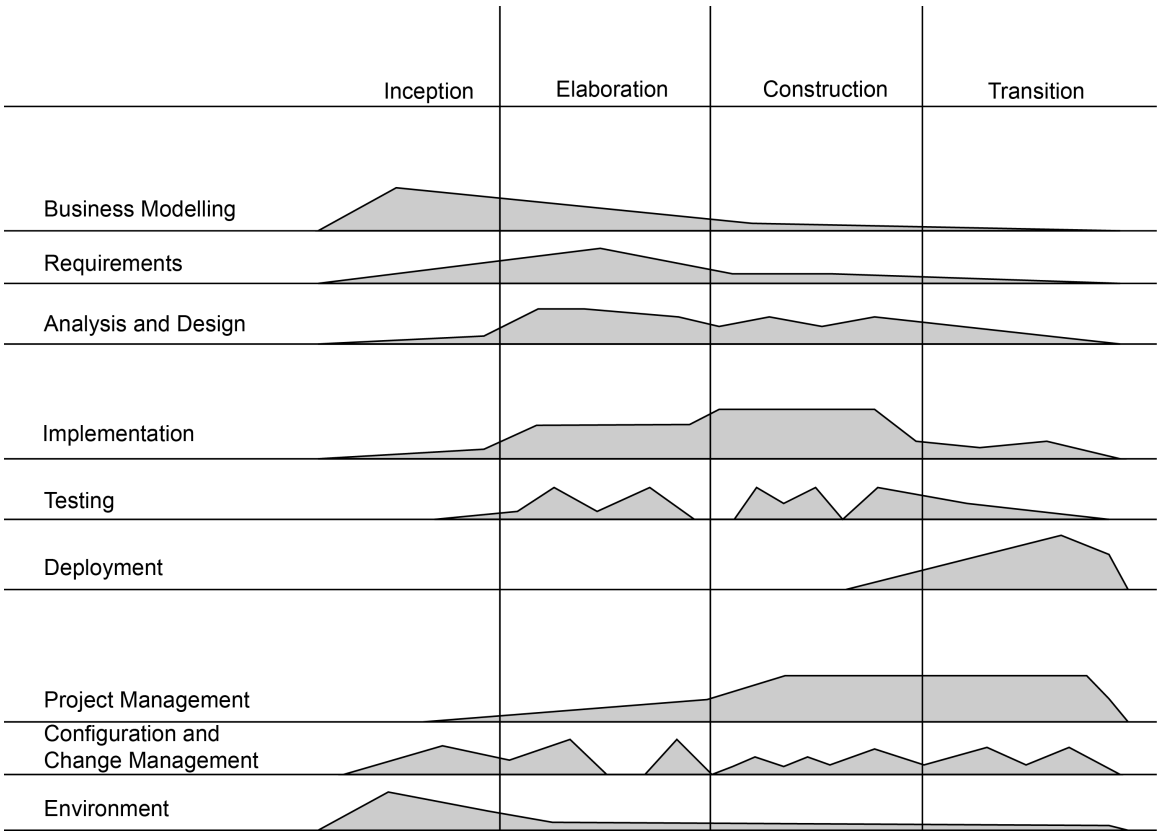

Figure 2.6: The Rational Unified Process Model

2.4.6 Agile Process Models

Agile software engineering methods were formalised by the introduction of the "Manifesto for Agile Software Development". The manifesto states (Beck et al., 2001a):

> "We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> — Individuals and interactions over processes and tools
> — Working software over comprehensive documentation
> — Customer collaboration over contract negotiation
> — Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more."

The Agile Manifesto seems to be contradictory in approach to the process models discussed in this section. Each of the process models is just that: a model of a

software engineering or development process. The Agile Manifesto, while conceding that process is important, deems individuals and interactions as more important.

The Agile Manifesto promotes twelve principles (Beck et al., 2001b).

1. The highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity — the art of maximizing the amount of work not done — is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Fowler (2005) proposes that Agile processes are both adaptive and people-oriented.

- Adaptive rather than predictive. All of the process models discussed previously in this section require a relatively large amount of design effort up-front, which implies that the work done in the construction phase of the project is predictable based on the design. This would be possible if the requirements of the customer for which the software is being built are static and change infrequently. This

assumption is known to be invalid (see earlier discussions on linear and iterative process models) and that means that the process of creating a software product should adapt to the inevitable change that a software project is subject to.

- People-oriented rather than process-oriented. Agile processes place emphasis on the people that will be involved in the project, and their specific and valuable skills, rather than the set of steps that generic resources are required to fulfil. The base of this approach is the idea that different people contribute differently to a project, making it impossible to assume that resources in a project are replaceable.

An example of an Agile Process Model is Extreme Programming (or XP). The following description of Extreme Programming is provided by Beck and Andres (2005).

> "Extreme Programming approaches software engineering by suggesting a set of core values, principles and practices. Values are the large-scale criteria we use to judge what we see, think and do, and are universal. Principles are domain-specific guidelines to life. Practices are evidence of values."

Extreme programming advocates the following core values:

- communication, in order to promote effective teamwork;
- simplicity, so as to avoid performing unnecessary (i.e. not required) work or communication;
- feedback, to ensure that the work being done is going in the right direction;
- courage, so that when things need to be done they are done; and
- respect, so that team members believe in themselves and other team members.

Extreme programming advocates the following core principles:

- humanity: because developers are people and have human needs (e.g. recognition and a sense of fulfillment), it is important to pay attention to keeping the human needs of the members of the team satisfied;
- economics: software should address business needs and deliver business value;

- mutual benefit: development activities should benefit the developer, developers working on the code in future and the customer;
- self-similarity: what may work in the small (or one context) may work in the large (or another context);
- improvement: focussing on getting things started quickly and improving them over time;
- diversity: using the diverse skills and approaches of team members for the best advantage of the team;
- reflection: thinking about how and why work is being done to improve the way things are done;
- flow: ensuring a continuous stream of value to the customer over time, rather than providing chunks of value at intervals;
- opportunity: seeing problems as opportunities for change;
- redundancy: providing more than one solution to a problem;
- failure: putting the focus on actually programming solutions, even if they fail, rather than spending too much time designing;
- quality: keep a focus on quality throughout the development activities;
- baby steps: provide small increments in functionality, which allows for shorter release cycles and more manageable work products; and
- accepted responsibility: developers must accept a task and own it and take pride in delivering it.

Extreme programming advocates the following primary practices:

- sit together: this promotes the team working together and communicating effectively;
- whole team: the team should contain all of the skills necessary to deliver the project;
- informative workspace: which gives team members a view of certain aspects of the project at all times (e.g. to do lists posted on the walls);
- energised work: focus on productive work rather than long hours;

- pair programming: this provides instant peer feedback that is integrated into the programming tasks of the project;
- stories: plan using units of customer-visible functionality (as opposed to rigid requirements) and base effort estimates on these stories;
- weekly cycle: plan work a week at a time and meet weekly to discuss it so that it is possible to quickly react to change;
- quarterly cycle: plan work a quarter at a time and meet quarterly so that alignment with bigger goals can be ensured;
- slack: plan in such a way that the tasks that are assigned can be moved to other cycles;
- ten-minute build: automatically build and test the whole system in ten minutes;
- continuous integration: integrate and test changes frequently (no more than 2 hours between integrations);
- test-first programming: create automated tests before writing code; and
- incremental design: build the design of the system up over the course of the project.

Extreme Programming is a paradigm that embraces change. The assumption that so many other software engineering processes make that it is important to have well-defined requirements before development work can start is not accepted by the Extreme Programming approach. Extreme Programming expects and embraces change in projects. There is no "big design up front", as the design is developed over the course of the project by continuously eliciting requirements from customers, developing small increments implementing these requirements and then getting feedback about whether the requirements were met or not. The values, principles and practices of Extreme Programming guide the development of a software project. In this way, Extreme Programming is not really a process model, but is more an approach. The result must be the same: a repeatable set of activities that yield an expected result.

The ideas of short release cycles, incremental design and the emphasis on the importance of testing are used in the approach discussed in Chapter 4.

This section described what software engineering is and what steps or processes are contained in a generic software engineering process. These steps are requirements analysis, design, construction, integration and testing. The work of Pressman (2005) was used to present a set of process models and an example of each process model was discussed. Each of the process models addresses the process steps that a generic software engineering process should contain (according to ISO 12207) and presents a different way of approaching these generic software engineering tasks.

The next section discusses software requirements, testing and defects and the relationship between them.

2.5 Software Requirements, Testing and Defects

As mentioned in Section 2.3.1, ISO9000 defines a *requirement* as *a need or expectation that is stated, generally implied or obligatory*. If one adopts the view of software engineering as the process of transforming requirements into software artifacts that are combined into a software product, then one can see requirements gathering as the starting point of a software engineering project and requirements fulfilment as the ending point of a software engineering project. At the onset of any software engineering project, there must be a set of requirements that define what is expected of the software once it is complete (with the exception of, for example, Extreme Programming, discussed in Section 2.4, where the team does not have all requirements at the beginning of the development effort, but it may have some). The end point of any software engineering project (software operation and maintenance notwithstanding) should be the verification that the software that has been produced is, in fact, what was required. So requirements are used as a definition of the product and artifacts, and a yardstick against which the product and artifacts are measured. The requirements of a software project are central to the determination of quality of the software produced. Requirements should be codified as a deliverable of a software project, and this is the case in the Software Engineering Process Models described in Section 2.4.

The IEEE (Institute for Electrical and Electronic Engineers) Computer Society makes the distinction between System Requirements and Software Requirements.

2.5.1 System Requirements Specification

A *system* is defined as (IEEE Computer Society, 1998a)

> "An interdependent group of people, objects, and procedures constituted to achieve defined objectives or some operational role by performing specified functions. A complete system includes all of the associated equipment, facilities, material, computer programs, firmware, technical documentation, services, and personnel required for operations and support to the degree necessary for self-sufficient use in its intended environment."

The System Requirement Specification (or SyRS in IEEE nomenclature) describes the capabilities that a system should exhibit and the requirements that the system should fulfil. The properties of a SyRS as a collection of requirements are as follows (IEEE Computer Society, 1998a):

- each requirement should be stated only once;
- requirements should not overlap;
- relationships between requirements must be defined to show how they interact in the system;
- all of the requirements identified by the customer and those needed to define the system must be stated;
- the requirements should be described consistently with respect to detail and presentation;
- the boundaries, scope and context of the requirements should be stated; and
- the specification of requirements should lend itself to change and revision management.

The SyRS should describe the system completely in terms of what the system is expected to do. This includes descriptions of interactions with other systems in the system's environment. The individual components of the system are not specified in detail, but their interactions and interfaces with other components are.

2.5.2 Software Requirements Specification

A *system* may contain a set of software components. In the SyRS, these components would have been described as parts of a bigger system. The expected reaction of the component to external interactions would have been specified in the SyRS too. The Software Requirements Specification (SRS in IEEE nomenclature) can be seen as complementary to the SyRS, describing in more detail components of the system; and certainly a required part of the overall specification of the system.

The SRS is (IEEE Computer Society, 1998b)

> "a specification for a particular software product, program, or set of programs that performs certain functions in a specific environment."

An SRS addresses the functionality, performance, external interfaces, attributes and design constraints. Properties of an SRS are:

- each requirement stated is one that will be delivered;
- each requirement can only be interpreted in one way;
- all significant requirements, particularly those requirements described in the SyRS are specified. The SRS must agree with, and be consistent with, the SyRS;
- all possible inputs to the software have been specified and the reaction of the software to these inputs has been specified;
- each requirement has an identifier that indicates the importance or stability of the requirement. Each requirement also has an identifier that is used to refer to the requirement;
- each requirement is verifiable. A requirement is verifiable if there is some process that can validate that a software product fulfills the requirement; and
- the SRS document lends itself to change and revision management.

According to these standards, the specification of the system and software should explicitly define what the produced system and software should do. These documents should be dynamic (in that they must be able to change over the course of the development life cycle) records of fact that show what should be delivered. Customers use these specifications to (among other things) impart their desires on the producers

of software, and the producers of software use these specifications to (among other things) know what to deliver. Both parties use these specifications to prove that what was delivered is what was required.

The entity for which the software is being produced (the customer or users) will have a set of requirements. These requirements come from some business need. One of the activities in the software engineering process is to gather the requirements of the customer or users. Once the requirements are defined, these requirements must be expressed as a set of specifications (the SyRS and/or SRS). In general, the requirements describe *what* the customer or user wants from the system, and the SyRS and SRS describe *how* these desires will be delivered upon. Requirements can be categorised broadly as functional or non-functional. (Paech and Kerkow, 2004) describes the difference as:

> The term "non-functional requirement" is used to delineate requirements focusing on "how good" software does something as opposed to the functional requirements, which focus on "what" the software does.

2.5.3 Software Testing

Software Testing is the process of verifying whether the produced software meets the requirements laid down. Testing should be handled as an explicit activity in the software project, as in the Software Engineering Process Models mentioned in Section 2.4. Each test verifies some portion of the overall set of requirements. The ideal outcome of the testing effort of a project is some proof that what was delivered is what was required (or what was defined in the specifications). Testing is the process of confirming adherence to requirements and hence is integral to the determination of quality.

The IEEE Standard for Software Verification and Validation (IEEE Computer Society, 2005) defines *validation* (of software) as

> "(A) The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.
> (B) The process of providing evidence that the software and its associated products satisfy system requirements allocated to software at the end of each life cycle activity, solve the right problem (e.g., correctly model physical laws, implement business rules, use the proper system assumptions), and satisfy intended use and user needs."

The same standard defines *verification* (of software) as

> "(A) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
> (B) The process of providing objective evidence that the software and its associated products conform to requirements (e.g., for correctness, completeness, consistency, accuracy) for all life cycle activities during each life cycle process (acquisition, supply, development, operation, and maintenance); satisfy standards, practices, and conventions during life cycle processes; and successfully complete each life cycle activity and satisfy all the criteria for initiating succeeding life cycle activities (e.g., building the software correctly)."

The IEEE Standard Glossary of Software Engineering Terminology (IEEE Computer Society, 1990) clarifies the definition of *verification* by adding that it is

> "Formal proof of program correctness."

CMMI for Development (Phillips, 2006) states that the purpose of the *validation* process in a software project is

> "to demonstrate that a product or product component fulfills its intended use when placed in its intended environment."

CMMI for Development also states that the purpose of the *verification* process in a software project is

> "to ensure that selected work products meet their specified requirements."

Validation can be seen as confirmation of requirements "in use": the product fulfils the desires of the user as they were stated. In other words, validation attempts to show that the user's requirements have been adhered to. Verification attempts to show that the user's requirements have been correctly translated into specifications.

CMMI for Development simplifies the difference between verification and validation:

> "verification ensures that *you built it right*; whereas, validation ensures that *you built the right thing*."

Verification of software implies a formal proof of correctness, and an approach such as Cleanroom (Mills et al., 1987) can be applied. Validation of software implies that some set of planned tests must be run against the produced software, or portions of the produced software. These tests are represented by a set of inputs, an expected result and an actual result. Software testing consists of the creation, execution and analysis of the tests that verify and validate the software. Just as the software and system specifications are required to indicate what it is that the customer wants and the developers should produce, the verification and validation effort must be specified. This specification is referred to as a Test Plan. A Test Plan defines which tests are to be run, by whom, and gives details about the timing of testing activities. A Test Plan contains a set of Test Cases. Each Test Case aims to verify and/or validate the adherence of some portion of the produced software to some specified requirement. The Test Case will define the context or environment that prevail at the time of testing, the input to use and the output to expect. If the actual output of the Test Case differs from the expected output, an *anomaly* is noted. This *anomaly* represents a deviation from the specified requirements of the software.

According to Pressman (2005)

> "Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project and the product."

There is a relationship between the Test Plan and the Software Requirement Specification (or System Requirement Specification). Test Cases in the Test Plan must prove that produced software fulfills requirements in the SRS or SyRS.

Testing, and its relationship to requirements forms a basis of the approach described in Chapter 4.

2.5.3.1 Testing strategies

There are 2 main testing strategies as defined by Abran et al. (2004), described below.

- Black-box testing is where tests are compiled using only the specification of the product; treating the software as a "black box" in that only the inputs and outputs are observed, not the actual processing. This testing strategy attempts to show that the software meets requirements. Because tests are created from the requirements, only the requirements will be tested, and situations such as

anomalous input may not be covered (because they may be implied and not explicitly stated in the requirements).

- White-box testing is where tests are compiled from the source code. The aim of white-box testing is to ensure that all execution paths of the code are tested. This testing strategy focuses on the code as opposed to the specification.

In order to achieve the most effective testing (i.e. ensuring that requirements are met and that running the software does not have unexpected results because of untested code paths) a combination of white- and black-box testing is suggested. Testing may be automated (i.e. a set of tasks run by some software) or manual (i.e. a set of tasks executed by a human being).

The approach described in Chapter 4 uses black-box testing: the functionality of the product is tested and not execution paths of code. The approach also uses a manual (i.e. human) approach to testing, where the tests are tasks performed by a human in order to verify the results.

2.5.3.2 Testing types

There are different types of tests that are run during the validation and verification phase. These descriptions come from the IEEE Standard for Software Verification and Validation (IEEE Computer Society, 2005), and are listed below.

- Component testing — Testing of individual hardware or software components or groups of related components. Sometimes referred to as *unit* testing.
- Integration testing — Testing in which software components, hardware components, or both, are combined and tested to evaluate the interaction between them.
- System testing — Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. Sometimes referred to as *end-to-end* testing.
- Acceptance testing — (A) Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system. (B) Formal testing conducted to enable a user, customer, or other authorized entity to determine whether to accept a system or component. Sometimes referred to as *user acceptance* testing.

Figure 2.7 shows the different types of tests that are used as the project progresses. In the earlier phases of the development life cycle, the components that are produced are tested. The components are tested in isolation or in small groups. The aim of this type of testing is to ensure that each component fulfills the requirements it was designed to fulfill. The design (or system or software requirements) must be used to derive the tests that are used to prove adherence to the design. Once each component is proven correct per specification, the components are integrated as required and tested together. While the emphasis of the component testing is to ensure each component is correct, the aim of integration testing is to ensure that the interaction between components is as required. System and acceptance testing essentially confirm the same thing: that when all of the components are integrated, they work per requirement internally (to the individual components) and integrate correctly to yield a system that adheres to all requirements. The main difference between the two is that acceptance testing involves the customer more explicitly as they need to make a decision about whether or not they will accept the product as complete.

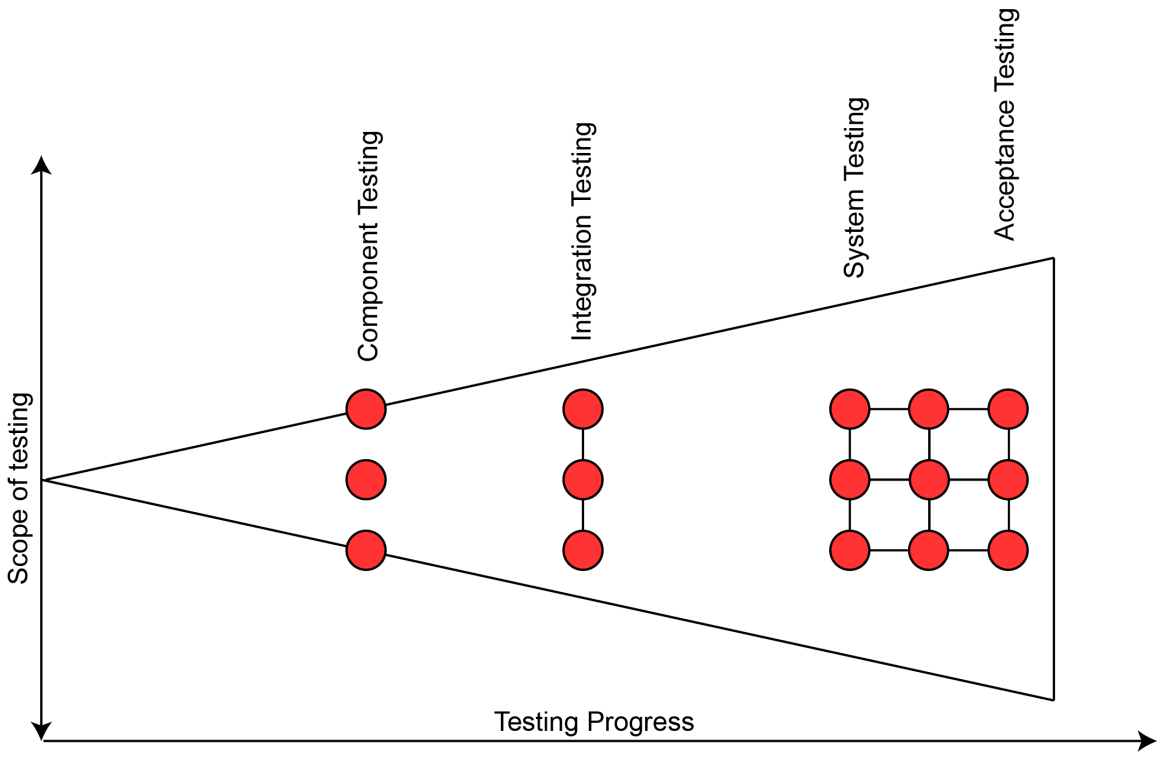

Figure 2.7: Testing scope as the project continues

2.5.4 Software Defects

In the context of a software product, a defect can be seen as a deviation from the required or specified result: a departure from the requirements. If the actual output of a Test Case differs from the expected output, an *anomaly* is noted. This *anomaly* represents a deviation from the specified requirements of the software. Thus, an anomaly and a defect amount to the same thing in this context.

The IEEE Standard Glossary of Software Engineering Terminology (IEEE Computer Society, 1990) provides a classification of errors.

- Error: The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.
- Fault: An incorrect step, process, or data definition. Also, a defect in a hardware device or component.
- Failure: An incorrect result. Also, The inability of a system or component to perform its required functions within specified performance requirements.
- Mistake: A human action that produces an incorrect result.

In a software project, all four types of error can arise and cause a deviation between the requirements and product.

Defects, and their relationship with tests and requirements form a basis for the approach described in Chapter 4.

2.5.5 The relationship between requirements, tests and defects

Requirements define what the customer wants. Tests verify that what was delivered meets the requirements. (Here, the assumption is that the requirements have been correctly translated into specifications upon which the software has been built and that the tests at least test the functional requirements). Defects are the result of deviations in the produced software from the requirements. This relationship is depicted in Figure 2.8.

Paraphrasing the definition of quality given in the ISO9000 (International Organization for Standardization, 2005) standards, the quality of a product is the degree to which the produced software (or parts of the produced software) meet the requirements. This entire definition of quality is encapsulated by the relationship depicted

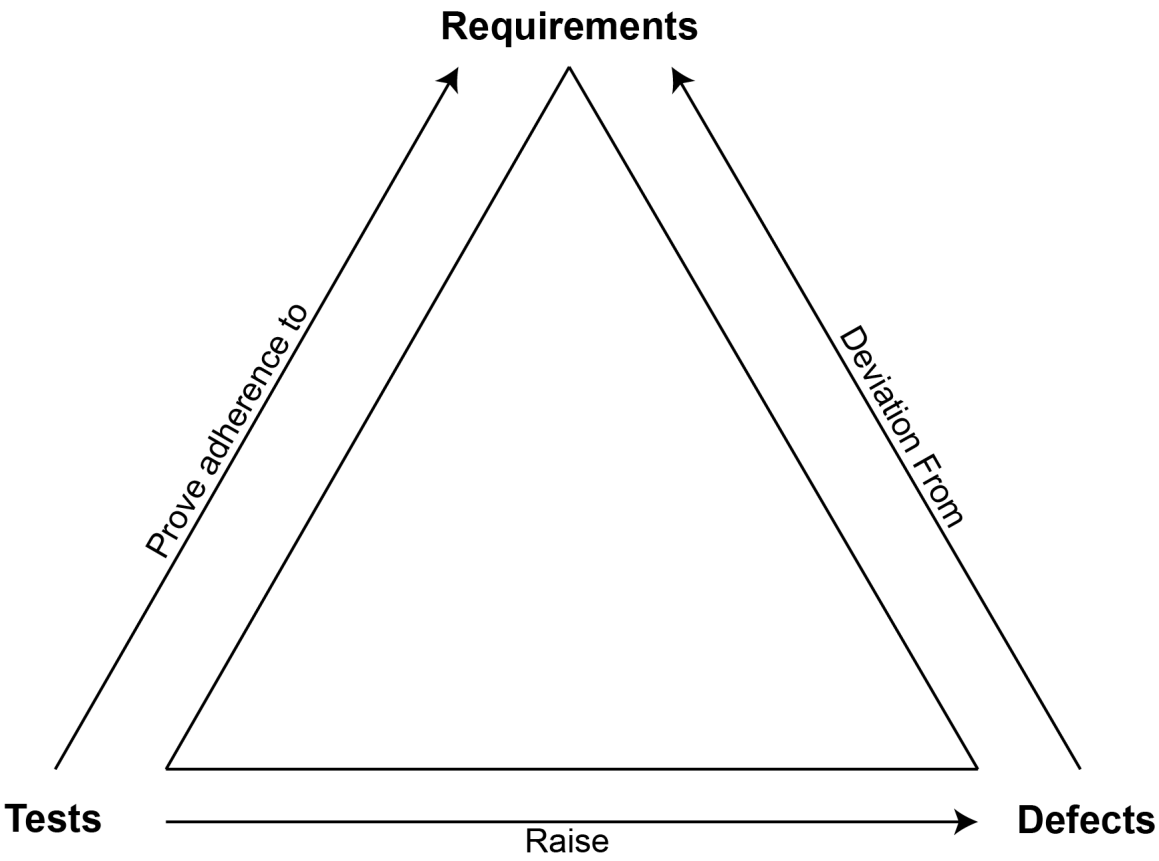

Figure 2.8: The relationship between requirements, tests and defects

in Figure 2.8. The requirements are present, the deviations from requirements are present (i.e. the defects) and the mechanism for deducing the difference between the two (i.e. the testing process) is present. Thus, it would appear, a very good view of the quality of software can be derived from study of this relationship. The discussion of Chapter 4 focuses on this relationship and uses this relationship to provide input data for deductions about quality.

This section described what is meant by software requirements, testing and defects and discussed the relationship between them. Requirements represent the features that the software is expected to provide. Testing provides a mechanism for validating that requirements are fulfilled. Defects represent the gap between what was required and what was delivered in the software (i.e. requirements and delivered product).

The next section discusses software metrics and software quality metrics.

2.6 Software and Software Quality Metrics

2.6.1 Metrics and Measurements

A *metric* can be defined as (IEEE Computer Society, 1990)

> "A quantitative measure of the degree to which a system, component, or process possesses a given attribute."

where a *measurement* (IEEE Computer Society, 2004) is

> "The act or process of assigning a number or category to an entity to describe an attribute of that entity. A figure, extent, or amount obtained by measuring."

When measuring some attribute of an object then, a metric is a way of representing an amount or extent of the attribute of the object.

Software metrics can be seen as tools for the measurement of the attributes of software. Measuring the characteristics or attributes of software can assist in gaining insight into the performance of the developers, testers, project managers and processes in a software engineering project. In software engineering projects, different stakeholders will want to know different things about the project, hence the need for different metrics. For example, the project sponsor may want to know the cost of the project; the developer may want to know how many defects the software has; the project manager may want to know how efficiently the project's resources are being used.

According to Fenton and Pfleeger (1998), software metrics can be grouped into three categories:

- process metrics, which measure the performance of the software development life cycle process;
- product metrics, which are used to measure the outputs of the development life cycle (e.g. requirements, source code, etc.); and
- project metrics, which are used to measure the available resources to the project.

Process metrics use measures of time, effort and cost in order to provide some feedback as to whether the process is being implemented efficiently and where improvement can be made (El-Wakil, 2004). This kind of metric is used by managers in order to ensure that the process used in the development lifecycle is appropriate. Product metrics are used by developers and their managers to provide insight into performance or quality of development lifecycle outputs such as requirements and source code. Project metrics (also known as resource metrics) are used by project managers and business owners in order to describe the performance of resources (human, hardware, etc.). The metrics categorisation of Fenton and Pfleeger (1998) explores the use of metrics across many aspects of software production (for example estimating cost, productivity and quality), while the focus of the ISO25000 family of standards is narrower (software quality).

Fenton and Pfleeger (1998) discuss software metrics in the context of the following applications of software measurement (among others).

2.6.1.1 Cost and effort estimation

Managers and project sponsors drive the need for this kind of software measurement. Measurements and estimates of cost and effort help managers act more proactively in assuring that projects are delivered on time and within budget. This information also provides feedback to those funding projects by informing them whether projects are going to be delivered on time and within budget. Having these kinds of measurements allows for the proactive monitoring of project progress. There are models that are used to estimate effort and duration of projects based on measurements performed on the software being produced. An examples of such a model is COCOMO (Boehm, 1981). Use of these kinds of models enables predictions and insight into the performance of the processes being used in the software project. The metrics commonly used are time and effort.

2.6.1.2 Productivity

This type of measurement aims to attain a clear idea of the efficiency of resources being used to deliver the product. These resources include human resources, the

teams of human resources in place and the tools being used to deliver the product. One way to determine productivity is to relate the amount of work performed to the time taken to perform the work. In terms of software development, this could be measured by the amount of code (measured, for example, in lines of code) and the number of person months (which is simply the number of people doing the work multiplied by the number of months taken to do the work) respectively.

2.6.1.3 Quality

The IEEE standard for a Software Quality Metrics Methodology (IEEE Computer Society, 2004) describes quality as

> "the degree to which software possesses a desired combination of quality attributes"

Furthermore, this standard asserts that

> "the use of software metrics within an organization or project is expected to have a beneficial effect by making software quality more visible."

A project, along with its explicit requirements of functionality, may have a set of quality requirements. Quality requirements are often referred to as non-functional requirements. These quality requirements are decided per project and are made up of a set of quality factors, which are measured by a metric. As part of the planning for a project, a set of quality factors are agreed upon, metrics used to measure the factors are identified and acceptable values for the factors are defined. At this point, it is worth noting that merely fulfilling these quality requirements is not enough to produce a product of high quality. These quality requirements should be specified and treated like any other requirement in the specification for the software. Chung and do Prado Leite (2009) suggest that specifying quality (or non-functional requirements) along with functional requirements assists in design decisions for the product.

One of the main aims of metrics in this context is to provide quantitative methods for the assessment of quality attributes of a software product. This is the core aim of the approach proposed in Chapter 4.

2.6.1.4 Reliability

Reliability is a predicted measure of how long software will run for without failure. One measure of reliability is Mean Time to Failure (MTTF). Because this measure

can only be calculated once a failure has actually occurred, reliability of software cannot be definitely known. However, software producers can use statistical techniques and experience from other projects to gain foresight into the expected reliability of software they produce. There are situations where organisations that use a software product need to have some idea of the reliability of the product. An example of such a situation is mission critical software where consequences of failures in the software may be dire. A customer could realistically expect a certain level of reliability in software they use. A customer uses a particular piece of software to fulfill some need or set of requirements, and the customer may see a failure in the software, no matter how reliable or infrequently realised, as a deviation from their requirements — albeit not necessarily explicitly stated requirements. The reliability of a piece of software is an intuitively good indicator of perceived quality. A customer would be more satisfied (and perceive higher quality) if the software they were using did not fail frequently.

2.6.1.5 Structure and complexity

Measures of the structure and complexity of software provide information about the development and maintenance effort applied to produce and maintain the product. An example of an approach to measuring complexity is McCabe's Cyclomatic Complexity Measure (McCabe, 1976), which depicts software at the code level as directed graphs (where nodes are program statements and edges are flows of control between these statements). A measure that is used to show that the testing effort expended is sufficient and effective is the Test Effectiveness Ratio. This measure relates the number of objects (here, objects refer to execution paths and similar constructs relating to structure) exercised one or more times to the total number of those objects.

Other measures that use structure and complexity information (in this case in relation to modularity) are coupling and cohesion (Stevens et al., 1974). Coupling is the degree of interdependence between modules and cohesion is the extent to which individual components are needed to perform some task. It is generally thought that it is a good design principle to aim for low coupling (where modules must communicate, they do so using simple parameters) and high cohesion (where a module performs a single well-defined function). According to (Taube-Schock et al., 2011), though, it is inevitable as systems get larger and larger, that there must exist some level of high coupling.

2.6.1.6 Process maturity

While the areas of software metrics discussed above deal with project and product metrics, an organisation should be able to measure its processes and the maturity or

effectiveness of those processes. This is done by developing a measurement framework based on process methodologies and frameworks like CMMI (see Section 2.3.2) or ISO9000 (see Section 2.3.1). Using CMMI, the maturity of the organisations processes are measured against a scale which runs from initial (least mature) to optimizing (most mature). Using the ISO9000 standards, an organisation can get some idea of what processes are in use in the organisation and how they relate to one another in order to analyse their effectiveness in the quality of outputs.

This section introduced the concepts of metrics and measurement, which are central to the work discussed in Chapter 3 and the approach proposed in Chapter 4. A categorisation of applications of metrics (as described by Fenton and Pfleeger (1998)) was discussed.

The next section discusses Software Configuration Management.

2.7 Software Configuration Management

As a software engineering project progresses, changes will be made to source code, documents and data (which are collectively known as the software configuration). This change needs to be managed, and in software projects, Software Configuration Management (SCM) is the term used to describe the management of this change. The management of change can be seen as a quality assurance process.

According to Pressman (2005), SCM can be seen as

> "the set of activities designed to manage change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made."

Here, the work products refer to the development artifacts, being the source code, documents and data (i.e. the software configuration) mentioned above. One or more

of these work products is also referred to as a Software Configuration Item. Software configuration items, though, are more than the development artifacts, and include tools used to produce the software product.

A Software Configuration Management System is a system (which encapsulates SCM processes) that facilitates the management of change to software products and the software configuration items that make up the products.

A SCM System has the following elements (Pressman, 2005):

- component elements;
- process elements;
- construction elements; and
- human elements.

Component elements are the tools and portions of the SCM software itself that manage the software configuration items. Process elements refer to the process of using the SCM to manage change. Construction elements refer to tools used to build products from software configuration items while ensuring the correct versions of these items are used to build the products. Human elements are the people using the SCM to manage change.

SCM systems use a repository or database to store information about the software configuration items. The repository is responsible for data integrity, so that software configuration items are reliably stored and related; information sharing, so that many developers in a team can work with the software configuration items effectively; tool integration, so that access to the software configuration items is catered for; data integration, so that many software configuration items can be stored; methodology enforcement, so that some sort of relationship can be observed between software configuration items; and document standardisation, which aids in the implementation of an approach for creating project documents.

The SCM system typically makes use of some sort of meta-model which determines how information is stored, accessed and secured in the repository as well as how the model can be extended to allow for extension of functionality.

The process of managing software configuration comprises a set of tasks that:

- identify the items that make up the software configuration, which classifies the software configuration items as *basic* or *aggregate* objects. Basic objects represent discrete units (in the context of the software product) such as the source code for a module. Aggregate objects are collections of basic objects;

- manage changes to these items, which involves distinctly identifying versions of software configurations and storing them in some database or repository;
- facilitate the construction of versions of the software product, which assists in building complete versions of a software product from the software configurations comprising that software product; and
- ensure that the quality of the software is maintained over time, which is done by implementing the process of change management.

The set of tools that an SCM system provides should allow developers to provide versioning (identifying the work products), dependency tracking and change management (establishing relationships among work products), requirements tracing, configuration management (managing different versions of the work products) and audit trails (auditing and reporting on the changes made).

The capabilities of the SCM, and its use in the development of software projects, are central to the approach proposed in Chapter 4.

This section described Software Configuration Management as the management of change to work products.

This chapter has set the context for this dissertation and the discussion of measuring quality using metrics. The concepts pertinent to the discussions to follow have been discussed so that their use in the further discussions of this dissertation are understood. The concepts covered were quality, software engineering, software requirements, testing and defects, and software (and software quality) metrics, and software configuration management.

The next chapter will investigate existing approaches to measuring quality of software using metrics.

Chapter 3

Measuring Quality using Software Metrics

This chapter discusses four approaches to measuring software quality using metrics. The literature discussed in this chapter was chosen to give a holistic and multi-faceted view of existing approaches.

The works of McCall et al. (1977) and Boehm et al. (1978) were chosen because theirs is early work on quality in software, and as such presents somewhat of a genesis for the field. They are also deemed important because of their continued citation in later works, such as Churcher and Irwin (2005), Østerlie and Wang (2006), Cantone and Donzelli (1998), Wagner (2007), Gross and Yu (2000), and Cysneiros and do Prado Leite (2004).

The work of Hyatt and Rosenburg (1996) was chosen because of the focus on the perspective of the project manager, as someone who is interested in ensuring that a project is successful. This provides somewhat of a "real-world" perspective to the approach.

The ISO25000 (International Organization for Standardization, 2007a) family of standards (also known as SQuaRE) was chosen because it is an industry standard and as such it provides a "suggested practice" approach to measuring software quality because the ISO develops standards that meet market needs, are based on global expert opinion, take input from various stakeholders and perspectives, and are based on consensus of these stakeholders (International Organization for Standardization, 2013).

The approaches have been discussed in chronological order. Each of the approaches is discussed in terms of how the approach suggests a software producer should infer software quality based on the measurements of metrics and how these metrics are

related to those quality characteristics that each approach deems desirable. The advantages and disadvantages of the approaches are then discussed.

3.1 McCall's Quality Factors

In their 1977 technical reports to the United States Air Force (McCall et al., 1977), McCall, et al. describe an approach to assessing software quality using metrics.

Software is described as having a set of *factors*.

A *factor* is defined as

> "A condition or characteristic which actively contributes to the quality of the software."

Each of these factors or characteristics are influenced by a set of *criteria*.

Criteria are defined as

> "Attributes of the software or software production process by which the factors can be judged and defined."

Presence or levels of the criteria assist in giving insight into the presence or level of characteristics of the software that influence the quality of the software. One criteria could be used to indicate more than one factor. The factors of software quality are evident in the use of the software, while the criteria are generally applicable to the production of software. McCall, et al. describe, in detail, their choices of factors and criteria and describe the complex inter-relationships between the factors and criteria. Table 3.1 shows the set of factors, and criteria that influence them, as defined by McCall, et al.

Metrics, then, are used to measure the criteria. McCall, et al. describe the metrics, provide a detailed set of metrics, and propose a mechanism for applying these metrics. Each metric is used to measure a particular criteria, in a specific phase of the software development lifecycle. Values are assigned to each metric such that the units of the metrics are expressed in terms of the ratio of the observed number of occurrences to the total possible number of occurrences. This method is used in the approach proposed in Chapter 4. Alternatively, the metrics are measured in binary form, which indicates the presence or lack of that which is being measured. This is an attempt to normalise the measurements and present them as quantitatively and objectively as possible.

An example metric according to (McCall et al., 1977) measures the cross reference relating modules to requirements. This metric attempts to measure the Traceability criterion, which contributes to the Correctness factor.

Having a model of quality that can be applied to any number of successive projects means that the same set of metrics apply, and a relative view of quality can be attained across software projects, i.e. metric performance or utility feedback is possible across projects. In addition, the metrics are defined as applicable to the different phases of the software project life cycle, which means that insight into quality performance can be gained at all phases of the project.

The list of factors and criteria described in Table 3.1 (and the related metrics) are based on the typical requirements and experiences of projects that were being run in the 1970's at the US Air Force. A set of factors and criteria were identified that represented the reality of projects in the Air Force. This does not preclude the applicability of this approach in other contexts, but using this approach may mean identifying factors, criteria and metrics that accurately represent and measure quality in the context of use.

The assessment of quality is done by completing a set of worksheets, which request values for metrics per criteria. This process could be a time-consuming task, which presents a natural limiting factor to using the approach regularly.

Because of the effort of performing the measurements and deriving a view of quality, the exercises may be done at a stage where feedback to the relevant phase of the current software project is infeasible.

The approach described in Chapter 4 aims to address the relative cost of gathering these metrics by gathering and calculating values automatically.

3.2 Boehm's Characteristics of Software Quality

In their book "Characteristics of Software Quality", Boehm et al. (1978) discuss an approach to analysing software quality based on a set of characteristics that are generally desirable for software to exhibit. The high-level characteristics of software quality are derived from three major concerns of a software user or purchaser listed below.

1. How well is the software being delivered usable as it is (without any changes)? This relates to the "As-is Utility" of the software.
2. How easy is the software to maintain? This relates to the "Maintainability" of the software.

Table 3.1: Factors and criteria influencing them, according to McCall, et al.

Factor	Description	Criteria
Correctness	Indicates how well the software fulfils users' needs	Traceability, consistency, completeness
Reliability	Indicates how well the software performs its function with precision	Error tolerance, consistency, accuracy, simplicity
Efficiency	Indicates the amount of computing resources required by the software to perform its function	Execution efficiency, storage efficiency
Integrity	Indicates how well the data of the software is protected from unauthorised access	Access control, access audit
Usability	Indicates how easily users can perform the functions of the software	Training, communicativeness, operability
Maintainability	Indicates the effort required to fix errors in the software (when it is operational, as opposed to being in the development phase)	Consistency, simplicity, conciseness, modularity, self-descriptiveness
Testability	Indicates the effort required to show that software performs the functions it should	Simplicity, modularity, instrumentation, self-descriptiveness
Flexibility	Indicates the effort required to modify the software	Modularity, generality, expandability, self-descriptiveness
Portability	Indicates the effort required to transfer the software from one hardware configuration to another	Modularity, self-descriptiveness, machine independence, software system independence
Reusability	Indicates the extent to which parts of the software can be used to build other software	Generality, modularity, software system independence, machine independence, self-descriptiveness
Interoperability	Indicates the effort required to integrate systems with one another	Modularity, communications commonality, data commonality

3. Can the software still be used if it is placed in a different environment? This relates to the "Portability" of the software.

These concerns and questions led Boehm et al. to iteratively develop a hierarchical set of characteristics that are generally desirable in software. The highest level of characteristics are As-is Utility, Maintainability and Portability (which represent the three questions above, respectively). These three high-level characteristics all influence and define the General Utility characteristic of the software. This means that the more a product is useful without modification, the easier it is to maintain, and the less the user or purchaser is locked into specific hardware or software environments, the more generally useful the user or purchaser will find the software.

As mentioned above, the characteristics model is hierarchical in nature, and each level of the hierarchy influences the level above it. The final and complete set of characteristics proposed by Boehm et al. is shown in Table 3.2.

Boehm et al. used these characteristics to attempt to define one or more metric(s) for each of the characteristics that would quantitatively show the degree to which the software displays that characteristic. The quality of the product, then, could be seen as a function of all of the identified metrics. Boehm, et al. discovered that there is no one metric which shows the quality rating, and that the application and combination of these metrics would change based on the needs and priorities of users. A set of metrics was defined in terms of the following list of criteria.

- The correlation with software quality, on a scale of high to low (this is a subjective measure).
- The potential benefit of the metric having a high score for a characteristic, where the scale ranges from the case where it is important for the metric to have a high score to where there is no real loss if the metric has a low score.
- The quantifiability (and cost) of automated evaluation of the metric, where the scale ranges from a metric that can be calculated using an automated algorithm to where the metric must be manually determined via software or a human.
- The feasibility and completeness of automated evaluations, where the scale ranges from the automated checker providing total evaluation of the metric to the automated checker providing inconclusive evaluation of the metric.

An initial set of metrics was proposed and then refined by Boehm et al. based on the above criteria. Boehm et al. assert that the major long-term benefits of using metrics for the evaluation of quality in software lies in the ability to predict properties

Table 3.2: The software characteristics of Boehm et al.

Top-level characteristic	**Intermediate characteristic**	**Low-level characteristic**
As-is utility	Reliability	Completeness Accuracy Consistency
	Efficiency	Device Efficiency Accessibility
	Human Engineering	Accessibility Communicativeness
Maintainability	Testability	Accessibility Communicativeness Structuredness Self-descriptiveness
	Understandability	Consistency Structuredness Self-descriptiveness Conciseness Legibility
	Modifiability	Augmentability Structuredness
Portability	Completeness Device-independence	

of future software that will be developed, and in the ability to develop tools to aid the reduction of the cost of producing software (i.e. meeting software requirements). The immediate benefits of applying metrics in this fashion are related to the reduction of errors in software (and the subsequent cost of fixing these errors). Boehm, et al. further describe a model for determining the cost saving that is achieved by applying metrics.

An example of a metric described by Boehm et al. is "Metric Number 1.12" which measures whether the code is readable or not. This metric contributes to the low-level characteristic of Legibility which contributes to the intermediate characteristic of Understandability, which contributes to the high level characteristic of Maintainability.

The approach described by Boehm et al. is dynamic in nature. The priorities and intended use of the software being produced cause greater emphasis to be placed on different characteristics (and hence metrics) in different environments, across software projects, and across users of the software. This means that effort may need to be expended in developing and refining the set of characteristics and metrics for each project and user. There is value placed on the automated evaluation of the metrics, but it is clear that if a metric cannot be evaluated automatically, it must be evaluated manually.

The approach proposed in Chapter 4 aims to provide value by using metrics that can be evaluated automatically from the source code and Software Configuration Management system.

3.3 Hyatt and Rosenburg's Software Quality Model

Hyatt and Rosenburg (1996) propose a software quality model based on the perspective of the project manager in a software project that produces software within a larger system. The requirements of the project manager that are addressed are that the software "works well enough" (which implies adherence to functional, performance and interface requirements as well as the qualitative requirements mentioned in ISO25010 (discussed in Section 2.6.1) and is "available when needed" (which means that the software must be a reliable part of the larger system). The project manager is interested in a *pragmatic* quality model which will assist him/her in most effectively managing the project of developing software. In order to do this, the model must address these two requirements (i.e. that the software "works well enough" and "is available when needed") and must identify the risks to these requirements not being met. This concept of a pragmatic approach is used in the approach proposed in Chapter 4.

Hyatt and Rosenburg identify areas of risk to the project and associate these risk areas with requirements of the quality model and metrics: the model and metrics must be able to quantify risk in each area, and derive a "total risk" for the project based on the sum of the individually quantified risk areas. Because it may not be possible to calculate a numeric representation of the overall risk, Hyatt and Rosenburg suggest a risk classification indicating a general level of risk (e.g. low, medium, high), as opposed to an exact number.

Table 3.3 shows the risk areas identified by Hyatt and Rosenburg.

Table 3.3: Hyatt and Rosenburg's risk areas

Risk area	Description
Correctness	The risk of errors in the software
Reliability	The risk that the software will fail often
Maintainability	The risk that there is not sufficient documentation to maintain the software
Reusability	The risk that subsequent projects may not be able to use portions of the software
Schedule	The risk of the project not being delivered on time

An example metric from (Hyatt and Rosenburg, 1996) is Number of Weak Phrases, which attempts to measure the ambiguity of the requirements. This contributes to the Ambiguity attribute, which contributes to the Requirements Quality Goal. This goal contributes to the Correctness risk area.

The model proposed by Hyatt and Rosenburg describes a set of quality goals: requirements quality, source code quality, implementation effectiveness and testing effectiveness. A set of attributes contributes to each of these goals, and each attribute in each set of attributes is measured by one or more metrics. Hyatt and Rosenburg's model prescribes these goals, attributes and metrics [1]

Each of the four goals in the model are related to project risk. If the requirements are of low quality, there is a risk that the software produced is perceived and/or delivered as low quality. If the source code is of low quality, there is a risk that the software does not work as required and/or contains errors, which will lead to a perception of low quality. If resources are not used effectively to perform the right tasks at the right time during implementation, there is a risk to completion of the software, which

[1] Hyatt and Rosenburg refer to the ISO25010 characteristics and sub-characteristics as goals and attributes, respectively.

could lead to the perception of low quality as other important tasks are hurried. If testing is not effective for a software product, defects and errors in the software may not be found and this may lead to the perception of low quality when the product is used.

The focus of Hyatt and Rosenburg's model is that of the risk to a project that would concern a project manager. The model gives project managers a good view of the status of a project. The status of the project, in this context, means the risk to successful completion (and hence quality) of the software project. As a consequence, the metrics used in this model are biased towards project managers.

3.4 ISO25000: SQuaRE

As discussed in Section 2.3.3, the ISO25000 (International Organization for Standardization, 2007a) family of standards (also called SQuaRE) is concerned with quality requirements specification, measurement and evaluation. ISO25010 (International Organization for Standardization, 2011b) describes a framework for quality models that categorize product quality into characteristics, which may be sub-divided into subcharacteristics and possibly further into sub-subcharacteristics. Characteristics or subcharacteristics may be broken down into quality properties. Quality measures are used to measure at any of the levels of this hierarchy. Quality measure elements provide input to the quality measures and measure attributes of resources consumed, the software product, the context of use, or the effects of the use of the product.

ISO25010 provides two models based on this framework: the product quality model, which aims to measure quality internally (during the development process) and externally (when the product is in operation) and the quality in use model, which aims to measure quality when a product is used by its intended audience. The product quality model provides eight characteristics and a total of thirty-one subcharacteristics. The quality in use model provides five characteristics and a total of nine subcharacteristics. An example of a characteristic defined by ISO25010 is Effectiveness, which the standard defines as the "accuracy and completeness with which users achieve specified goals". For a description of these characteristics and sub-characteristics, please see Section 2.3.3. Since there are a large number of characteristics and sub-characteristics, the model allows for tailoring of the set of characteristics and sub-characteristics to fit the purpose of the organisation and project.

ISO25020 (International Organization for Standardization, 2008b) provides guidelines for defining measures for the quality model. These guidelines are listed below.

- When selecting measures to apply, document the reasons for selecting the measures. In SQuaRE, the reasons may be to address the needs of the stakeholders.

- When modifying existing — or building new — measures, document how the measure relates to the quality model and also how it is constructed from quality measure elements.
- Document the measures used and the measure elements that constitute the measures, including the criteria for selection, the measurement function and the measurement method.

ISO25020 defines a reference model for software product quality measurement called SPQM-RM. SPQM-RM describes the relationship between the quality model (as defined by ISO25010) and the quality measures that measure the characteristics and sub-characteristics that the model defines. Quality measures are made up of Quality measure elements. Figure 3.1 depicts SPQM-RM.

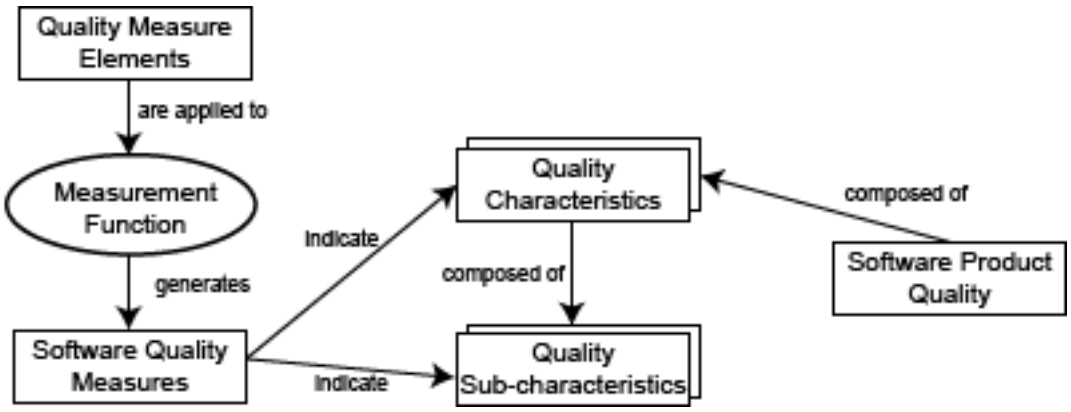

Figure 3.1: ISO25020 SPQM-RM

ISO25021 (International Organization for Standardization, 2008c) provides an initial set of quality measure elements that implementors of SQuaRE can use. The yet-to-be-published standards ISO25022, ISO25023 and ISO25024 provide detail regarding quality measurement of internal, external and in-use quality measurement, respectively, and will use the elements discussed in ISO25021 as a basis.

ISO25021 categorises the quality measure elements. An example category is Number of Functions, and an example quality measure element is Number of functions described in requirement specifications. Each quality measure element is defined in terms of the input source of information, the documentation required, the scale, focus and method of measurement, and the characteristic (from ISO25010) to which the quality measure element contributes.

In order to implement SQuaRE, a software producer would need to define the characteristics and sub-characteristics that they wish to measure, define the measures they wish to use, and define the measure elements that constitute those measures.

The SQuaRE approach provides software acquirers, suppliers, developers, operators and maintainers with a process that can be followed to evaluate the quality of a software product. The evaluation can be performed at various stages of the development lifecycle (being applicable to interim as well as end products). The evaluation gives feedback to managers and developers alike, and this gives them a chance to proactively address quality issues. Because the characteristics, sub-characteristics and measures used depend on the purpose of the evaluation, it may be necessary to redefine the set of characteristics, sub-characteristics and attributes for each evaluation. The same is true for the measures used. This may present a significant effort for the evaluators of software quality.

The approach described in Chapter 4 uses the ideas of executing the evaluation with a set of selected metrics as discussed here.

3.5 Discussion

As with the standards discussed in Section 2.3, the approaches discussed here each have their supporters and detractors. Kitchenham and Pfleeger (1996) criticise McCall's model for being too subjective, and ISO9126 (upon which SQuaRE is built) for not providing enough detail about the implementation and rather focussing on the "what" than the "how". SQuaRE does address this by providing reference models and sample measure elements, which helps implementors. Hyatt and Rosenberg's model is naturally limited by the fact that it focusses only on one stakeholder's perspective. ISO 14598 has been criticised because it is so closely related to ISO9126, and a unified standard would cause less confusion and promote adoption (Azume, 2001), and this has been addressed to some extent by SQuaRE.

Software quality models have been criticised for lacking standardisation in their definition which leads to an unclear understanding of their use. Deissenboeck et al. (2009) provide a set of requirements for quality models in the hope that they are more user-friendly and more widely used.

All of the approaches discussed in this chapter appear to agree that it is possible to use metrics to gain some insight into the quality of a software product, at various stages of development, as well as for the finished product. The approaches also all suggest a hierarchical view of characteristics, sub-characteristics and properties (although naming may differ by approach), with the exception of the approach of McCall et al. which has a 2-level hierarchy. It is generally accepted that the characteristics (which are what is seen as desirable in a software product) are influenced by the sub-characteristics, which are influenced by properties, which are measured by metrics or measures. Table 3.4 summarises the terminology of the approaches discussed here.

Table 3.4: Terminology of software quality measurement approaches

Approach	High-level characteristic	Intermediate characteristic	Low-level characteristic
SQuaRE	Characteristic	Sub-characteristic	Property
McCall et al.	Factor	N/A	Criteria
Hyatt and Rosenburg	Risk Area	Goal	Attribute
Boehm et al.	High-level characteristic	Intermediate characteristic	Low-level characteristic

In all of the models, there is the need to define a set of characteristics, sub-characteristics and metrics (called measures in SQuaRE). It may be necessary to refine these sets for each product and even at different project lifecycle stages of the same project. This may present a significant effort and a natural resistance to performing such activities which may affect timelines and budgets of projects. Hyatt and Rosenburg state "It is very difficult to convince [Goddard Space Flight Center] project managers to dedicate part of a mission's budget for a metrics program" (Hyatt and Rosenburg, 1996). Boehm et al. (1978) go to great lengths to present a model for calculating the cost of such an effort, and give an example to show that if metrics can be applied for less than a certain percentage of the cost of conventional testing, they would be worthwhile. The effort to define these characteristics is a necessary effort though, if organisations wish to achieve a view of software quality using these approaches. The effort required to define these characteristics could be larger the first time they're created, and reduced over time in the case where an organisation is producing software for similar contexts of use.

The sets of high-level characteristics or intermediate characteristics (where appropriate) defined by each approach are shown in Table 3.5. This highlights that there are some challenges when comparing these approaches.

- There is considerable overlap of the defined characteristics between the approaches.
- The mapping of intermediate characteristics may differ between approaches. For example in SQuaRE, Portability is made up of adaptability, installability and replaceability, while in the approach of Boehm et al., Portability is comprised of completeness and device-independence.

- High-level characteristics in one model are not necessarily high-level characteristics in another model. For example, in the model of McCall et al. integrity is a high-level characteristic, while in SQuaRE, it is a sub-characteristic (of Security).
- Each approach uses a different naming convention for the components of software quality. In light of this, García et al. (2006) propose a consistent naming convention for software measurement.

In addition to defining the characteristics, sub-characteristics and properties of the software quality model that a software producer intends to use, it is necessary to expend effort on actually collecting and evaluating the measurements attained. Of all of the approaches discussed in this chapter, only Boehm, et al. make mention that automation of metric collection and evaluation is desirable. Boehm et al. place emphasis on automated evaluation and collection by indicating that it is a desirable metric attribute. The automated collection and evaluation of measurements is indeed a benefit, as this is a repeated task that may need to be performed many times over the course of a project. There is certainly benefit to regular measurements, and if these measurements were automated, it would represent less of a barrier to implement them.

SQuaRE makes a distinction between the *Quality Requirements* of a software product and the *Requirements* of a software product. The same is true for Hyatt and Rosenburg, who refer to Quality Goals instead of Quality Requirements. The requirements are those things specified and agreed upon by the customer as desirable in the software. The quality requirements are those things specified by the producer of software as being desirable in the software. The customer desires certain functionality and capability from the software so that they may derive utility from the software and "get what they pay for". The software producer desires certain characteristics of the software to be evident so that the software producer can show that the software exhibits certain levels of quality. Using, and developing, software in accordance with a quality specification, enables software producers to ensure consistent quality across projects, which is something promoted by SQuaRE (see the descriptions of process quality feedback in Section 2.3.3).

Something that is not explicitly mentioned by any of the approaches discussed in this chapter is the software producer's choice of Software Development Methodology (see Section 2.4), and how this effects the measurement of quality. All of the approaches seem to imply a software development methodology that begins the software development process with the completion of requirements specifications. These specifications are used to define quality characteristics, sub-characteristics and properties. In the event that the software producer is using an Agile methodology (see Section 2.4.6),

Table 3.5: Comparison of quality characteristics per approach

High-level characteristic	SQuaRE	McCall et al.	Hyatt and Rosenberg	Boehm et al.
Compatability	Y			
Context coverage	Y			
Correctness		Y	Y	
Effectiveness	Y			
Efficiency	Y	Y		Y
Flexibility		Y		
Freedom from risk	Y			
Functional suitability	Y			
Human-Engineering				Y
Implementation Effectiveness			Y	
Integrity		Y		
Interoperability		Y		
Maintainability	Y	Y	Y	Y
Modifiability				Y
Performance efficiency	Y			
Portability	Y			Y
Product Quality			Y	
Reliability	Y	Y	Y	Y
Requirements Quality			Y	
Reusability		Y	Y	
Satisfaction	Y			
Schedule			Y	
Security	Y			
Testability		Y		Y
Testing Effectiveness			Y	
Understandability				Y
Usability	Y	Y		

there is no big upfront design, but there is a smaller set of requirements that change over time from which these characteristics, etc. can be defined. This may mean that a generic set of characteristics, etc. must be used (which is good because of the time saving due to not having to re-define them based on requirements; and bad because there may not be as good a view of quality as there would be if context-specific characteristics, etc. had been defined).

This chapter discussed four approaches to using software metrics to provide insight into the quality of software. The major advantage of the approaches was that there seemed to be agreement that it is indeed possible (and desirable) to measure quality using metrics. The major disadvantage was that there could be significant effort required to (a) redefine the characteristics, sub-characteristics, properties and metrics or measures to be used to evaluate quality and (b) collect and evaluate the measurements obtained.

The next chapter will describe a new approach, using the material discussed in all previous chapters, to providing insight into software quality, without the need to expend massive amounts of effort on manual collection and evaluation of measurements.

Chapter 4

A pragmatic approach to measuring software quality

This chapter describes an approach to providing insight into the quality of a software product using information stored for each software configuration item in a software configuration management (SCM) repository. Many software engineering methodologies suggest the use of an SCM system in order to manage the inevitable change associated with software projects (for example the Rational Unified Process discussed in Section 2.4.5). Because developers should be using an SCM during the software project, the SCM is an integral part of the process of developing software. Extending the usage of the SCM to include storing information about quality may help to provide insight into quality without the need to add many extra processes or systems to the developer's effort or toolkit.

The approach proposed here suggests that information pertaining to the requirements, tests and defects of the project and source code be stored in the repository along with the source code and other files. The approach suggests methods and mechanisms that can be used to store and analyse this information (including an equation illustrating the perceived quality based on this information) in order to provide insight into the quality of the product that the contents of the repository represents.

This chapter also describes a reference implementation of the approach using Subversion (Collins-Sussman et al., 2004) as the SCM repository. This reference implementation is called *Metaversion*. It is important to note that the concept and approach could be implemented using any SCM system that supports versioned metadata.

4.1 A pragmatic approach

The Paperback Oxford English Dictionary defines *Pragmatic* as (Soanes et al., 2006)

> "dealing with things in a sensible and realistic way."

And *pragmatism* is defined as

> "a realistic and sensible attitude or approach to something."

Another way of describing pragmatism is where the solution that is used in a particular context is one that works and makes sense. A pragmatic solution is one that satisfies the constraints and requirements of a problem even if it is not an optimal solution. In some ways, the concept of pragmatism is similar to the concept of "satisficing", described by Simon (1956), as

> "a satisficing strategy may often be (near) optimal if the costs of the decision-making process itself, such as the cost of obtaining complete information, are considered in the outcome calculus."

In cases where finding an optimal solution requires expending time and effort, a pragmatic solution offers the opportunity to save both.

The concept described in this chapter provides a pragmatic (and satisficing) approach to the measurement of software quality because it uses information (tests, defects and requirements) and processes (the use of an SCM) that are already in place to provide an insight into quality. Examples of the processes used in developing software are discussed in Section 2.4. The approach to measuring software quality described here attempts to use the metrics *number of defects*, *number of test* and *requirements addressed* related to each part of the software product (i.e. the source code of the software) to provide inputs to an equation that may be used to provide insight into quality.

4.2 Assumptions

A software engineering project can be defined as a project undertaken to provide some software product that fulfills some requirements of a user or set of users. One of the indicators of quality in a software engineering project is the degree to which the produced software fulfills the stated requirements of the user.

A software project consists of a set of deliverables. Among these deliverables are code and documentation. The delivered code and documentation are created from some source code (and documentation) that are developed over the course of the project by software developers. The source code is represented by a set of files. Over the course of the project, these source files will be created, modified and tested until they fulfill the stated requirements.

Most software engineering methodologies require a testing phase (see examples in Section 2.4), so at certain points of the software development life cycle, tests will be run against an interim software product, or *release*. The execution of these tests may uncover differences between the stated requirements and what the interim product actually delivers. These differences will be handled as defects. Successive phases or iterations of the development life cycle, then, will attempt to address these defects so that there is no deviation between the product and the requirements (i.e. there are no defects).

The assumptions used in this approach are listed below.

- There is a set of requirements indicating the requirements that the developers of the software need to fulfill. The requirements are easily identifiable and can be enumerated. The requirements may take the form of a System Requirements Specification (SyRS) or Software Requirements Specification (SRS) as defined in Section 2.5.2. Note that this assumption of the existence of requirements does not preclude using an Agile methodology (see Section 2.4) because Agile approaches do have requirements, they are just less numerous at first and gathered as the project progresses. The requirements can be functional or non-functional, but this approach assumes that (1) the requirements (functional and non-functional) have been stated in this set of requirements and (2) all of the stated requirements can be verified using some test.

- There is a set of source files that are developed to fulfill the requirements of the software product. Each source file contributes to the fulfillment of zero or more requirements. Some source files may provide helper functions such as string formatting and not be directly involved in fulfilling requirements. A requirement may be fulfilled by a single source file or a number of source files.

- There is a set of tests that are run against a set of source files to verify that there is no deviation between what is delivered and what is required (i.e. there are no defects). A test exposes only one defect and a defect is raised by only one test. This does imply some constraint on how tests are designed.

- Each requirement will be verified by one or more test(s). Because source files

are associated with requirements, this implies that a set of tests is associated with a set of source files — the cardinality of the relationship being unrestricted.

- There is a set of defects, which represent the deviation between delivered product and stated requirements. Each defect is found as a result of a test execution. Each source file may be associated with zero or more defect(s) and each defect may be associated with one or more file(s). As noted by Edsger Dijkstra in (Naur and B. Randell, 1961), "Program testing can be used to show the presence of bugs, but never to show their absence", and the approach proposed here only deals with those defects that are observable as a result of running tests on the software. If there are latent defects that are not uncovered by testing (for example, because the testing was not rigorous enough), then these defects would not be considered in this approach.

4.3 Metrics and measures used

Based on the assumptions described above, this approach uses the metrics *requirements*, *tests* and *defects* to measure the quality of a software product. This approach proposes that one could infer information about the quality of the software product by considering these metrics.

The fewer defects a product has, the less deviation from the stated requirements, the more probable it is that the customer will be satisfied; the higher the quality of the product.

The number of tests may or may not imply information about the quality of a product: having more tests may or may not imply that the producer of software may be more accurate in whatever claims it makes regarding quality. What is perhaps more direct in assessing quality is the proportion of tests that pass to the total number of tests. This indicates that of all of the tests that are written, some percentage of those tests pass, which implies (if the assumption is made that a test confirms agreement with a requirement) that at least some of the requirements are fulfilled. Combining these two pieces of information, the combination of the overall number of tests in relation to the source files and the proportion of tests that pass could imply more information about quality. If every source file were to be represented by a test and a high proportion of those tests passed, some conclusion about the quality of the product being high might be drawn.

Similarly, the number of requirements does not directly imply anything about the quality of the product. However, what is of importance is the proportion of the requirements that are met to the total number of requirements. The aim is to meet

all of the customer requirements, and hence produce a quality output. The difference between all of the requirements that have been specified and those that have been met (and shown to be met using a test) gives an indication of the quality of the product. In general, the requirements that are to be met include functional and non-functional requirements as well as specified and implied requirements. However, in order to ascertain the total number of requirements, it is important that they be enumerable. Hence, as far as possible, implied requirements should be stated explicitly.

4.4 An equation for measuring quality

Given the relationship between tests, defects and requirements discussed in this and previous chapters, and the idea that the quality of a software product is the degree to which it fulfills its stated requirements, Equation 4.3 provides a way of assessing the quality of a software product. Equation 4.3, named the Product Quality Indicator, calculates a measure of quality based on a weighted arithmetic mean of the proportion of requirements that have been met to those that have not (Equation 4.1) and the proportion of tests that have passed to those that have not (Equation 4.2). These equations do not take into account that certain requirements are more important than others, and that certain requirements require more effort than others to fulfill; these equations only count the number of requirements and tests and do not assign any importance or weighting to them individually.

4.4.1 The Completeness Indicator

The Completeness Indicator (Equation 4.1) is defined as the proportion of requirements met to the total number of requirements specified.

$$R = \frac{R_{met}}{R_{total}} \tag{4.1}$$

where R_{met} is the number of requirements claimed to be met and R_{total} is the total number of requirements. The concept of using the proportion of total requirements to the proportion of errors is also discussed in Wyatt et al. (2003). Michael et al. (2001) and Richter (1999) also discuss using the proportion of requirements met.

The proportion of the number of requirements met to the total number of requirements, R, gives an indication of the degree to which the software product fulfills its requirements. The proportion does not give any insight into the requirements that are not yet met though. These unmet requirements may be unmet because of defects

or because the requirement simply has not been implemented yet (for projects still in the development phase). It is clear to see that (in an ideal world) as a project nears the end of its development phase, R would tend towards 1 because the number of requirements that are met should be the same as the number of requirements in total by the end of the development effort.

4.4.2 The Completeness Confidence Indicator

The Completeness Confidence Indicator (Equation 4.2) is defined as the proportion of tests that pass to the total number of tests specified.

$$T = \frac{T_{run} - D}{T_{total}} \tag{4.2}$$

where T_{run} is the number of tests run (pass or fail), T_{total} is the total number of tests defined, and D is the total number of defects raised by the tests that have run. As stated in the assumptions earlier in this chapter, each test can give rise to zero or one defects. The concept of using the proportion of tests passing to failing is discussed in Vetro et al. (2011) and Cohn and Ford (2003).

The difference between the number of tests run and the number of defects raised shows how many of the tests that have been run have passed (i.e. without raising a defect). This is true because of the assumption that a test gives rise to only one defect and a defect is raised by only one test stated earlier. As the development effort progresses, the number of tests run T_{run} should increase (as more requirements are stated to be met by the developers) towards the limit of the total number of tests T_{total}. In addition, the number of defects should decrease towards 0, meaning that the value of T would tend towards 1.

4.4.3 The Product Quality Indicator

So the Product Quality Indicator Q, then, is

$$Q = \alpha \times R + (1 - \alpha) \times T \tag{4.3}$$

where α is the weight of the value of R and $0 < \alpha < 0.5$.

As indicated in Equation 4.3, Q is calculated as a linear combination of R and T where the sum of the weights (or co-efficients) is 1. The reason for this is so that different importance can be placed on each of R and T. For example, a relatively low value of α would ensure that Q remains realistic in a context where a high proportion

of requirements have been met but a low proportion of tests have passed. However, having a high proportion of requirements met with a low proportion of tests passing should not result in a high value of Q. Theoretically the value of α is between 0 and 1. Here, however, α is constrained between 0 and 0.5, so that the value of T is given a higher weight than the value of R in the calculation of Q because the weighted value of R will always be less than 0.5. The values of R and T range between 0 and 1 and hence Q ranges between 0 and 1.

Alternative methods of calculating Q were considered and they are listed below.

Equation 4.4 calculates Q as an un-weighted arithmetic mean, or average, of R and T. Both Equation 4.4 and Equation 4.3 rely on the same underlying statistical model (of arithmetic mean), but Equation 4.4 uses a value of 0.5 for α in Equation 4.3. As discussed above, T should have a higher weight than R, so an arithmetic mean is insufficient to express the Product Quality Indicator. In the extreme case where R is 1 and T is 0, an arithmetic mean would yield Q of 0.5. This is not a true reflection of the quality of the product. When R is 1, it implies that all requirements are claimed to be met. The value of T being 0, however, implies that no tests have passed successfully. If every test was failing then the quality of the product would be considered low. If no tests have been run, then the value of Q can not be based only on the proportion of requirements that have been met without decreasing its value with a weighting as indicated in Equation 4.3.

$$Q_1 = \frac{R + T}{2} \tag{4.4}$$

Equation 4.5 calculates Q as a geometric mean of R and T. In contrast to Equation 4.4 above, if T was 0, Q would be 0. This implies that the product has the least possible quality until at least one test passes. This is also not a true reflection of the quality of the product. As a project progresses, the value of Q should increase towards 1, indicating that work is progressing towards achieving maximum quality. If all of the requirements were met (i.e. R was 1), then the quality of the product should not be 0 even if no tests had passed, because that gives the impression that no work has been done on the project.

$$Q_2 = \sqrt{R \times T} \tag{4.5}$$

Equation 4.6 considers a vector (R, T) and calculates Q as the magnitude of that vector. Considering a vector has the benefit of providing a Product Quality Indicator that also describes the completeness and completeness confidence and hence provides

more information. However, considering Q as the magnitude of the vector has the disadvantage that the value of Q would always be equal to or greater than the value of R (and also greater than the value of T because R is greater than T). This means that the quality of the product is determined mostly by the proportion of requirements fulfilled, without giving proper weight to the proportion of tests that pass.

$$Q_3 = \|(R, T)\| = \sqrt{R^2 + T^2} \tag{4.6}$$

Mordal-Manet et al. (2011) criticize the use of averages in the calculation of software quality metrics, suggesting that a simple average may not convey the standard deviation of the population and that a weighted average may lack confidence in the result. In their Squale model, they propose to use a discrete scale for manual measures and an aggregation function for measures that can be attained automatically. There are two steps to calculating the metric automatically. First, a weighting function of the form $g(IM) = \lambda^{-IM}$ is applied to each individual mark or score, where IM is the individual mark or score of each measure and λ is a constant associated with the degree of tolerance for low values of IM (high tolerance is represented by a low λ and low tolerance is indicated by a high λ). Secondly, the weighted results are added together and scaled logarithmically so as to normalize it into the range of the individual marks or scores, yielding the equation $-\log_\lambda\left(\frac{\sum_1^n \lambda^{-IM_n}}{n}\right)$ where n is the number of marks or scores.

In this case, since the value of Q is intended to be collected automatically, a Squale function can be applied to calculate it. A value of 2 is chosen for λ since there is a high tolerance for low values of R and T. Equation 4.7 calculates Q according to this function.

$$Q_4 = -\log_2\left(\frac{2^{-R} + 2^{-T}}{2}\right) \tag{4.7}$$

These equations have been aimed at the software product as a whole. The same concept, though, can be applied to each source file involved in a software product. Each source file may have a set of tests, requirements and defects associated with it.

A test would usually be run to confirm some functional or non-functional aspect of the software product or some portion of the software product. A number of source files may contribute toward that functional or non-functional aspect, and a number of tests may test the contribution of a source file. This many-to-many relationship between source files and tests means that it is possible to identify which tests affect which source files. In the case of unit testing, the relationship of source files to tests is more clear because the unit test is created to test some unit (perhaps even one file) of the software product.

A test would either pass or fail. If a test fails, a defect is raised. Because the test is evidence of adherence to the requirements and a defect implies the failure of a test, a defect shows a deviation from the requirements. When a test associated with a source file fails, a defect is then associated with the source file. This association of defect with source file may require some intermediate work such as analysing the root cause of the defect and pin-pointing exactly which source file(s) cause(s) a defect. There may be more than one source file responsible for a defect and one source file could cause more than one defect.

When a developer creates a source file, he/she does so for a reason: there is some need for the source file to contribute toward meeting the requirements of the software product as a whole. This implies that when a source file is created, it is known which requirements the source file is aiming to fulfill. It is not necessarily the case that each source file addresses one or more requirements. Each source file may be part of at least one set of source files addressing specific requirements. The requirements that a source file assists in addressing must be proven to be fulfilled (or not) by tests (or defects).

So for any source file, the same equation for Q as proposed in Equation 4.3 can be applied.

4.5 Analysis of the approach

The attempt to measure software quality is by no means a new undertaking, with work from Boehm et al. (1978) being done more than 30 years ago. The approaches discussed in Chapter 3 all suggest a view that metrics could be used to provide insight into the quality of software. The major disadvantage of the approaches discussed is that there could be significant effort required to (a) define or redefine the characteristics, sub-characteristics, attributes and metrics to be used to evaluate quality and (b) collect and evaluate the measurements obtained.

Similar work includes (Barkmann et al., 2009) and (Gousios and Spinellis, 2009). Both approaches propose automated evaluation of metrics based on source code using a Software Configuration Management system as the repository. Barkmann et al. (2009) rely on source code from large number of external open source code repositories, although the approach could theoretically be applied to a software producer's local repository. This approach attempts to validate which software metrics to apply based on an analysis of the source code of these open source projects. Gousios and Spinellis (2009) built a complex software system that replicates external data (being source code from the code repository, email messages and bugs) into local repositories. It also provides a framework that would allow a software producer to write

code to calculate any number of metrics based on information stored in these data stores. While both approaches provide a method for software producers to analyse the software using metrics, the focus is not on quality specifically, but rather collecting and analyzing metrics in an automated manner. Other works that use Software Configuration Management systems to extract information include (Pérez et al., 2012), which uses GIT (Swicegood, 2008) repositories to examine the evolution of projects and developers, and (Poncin et al., 2011), which uses Subversion (Collins-Sussman et al., 2004) and other systems used in the development process (such as bug tracking software and mail archives) to examine the activities of developers.

There are also solutions available that integrate the SCM and bug-tracking systems to co-ordinate the mapping of defects to source code. Examples of such solutions are the integration between IBM Rational ClearCase (SCM software) and IBM Rational ClearQuest (bug tracking software) (Wahli et al., 2004) — though this integration is being discontinued at the time of writing — and the open source project Scmbug, which integrates Bugzilla (bug tracking software) and Subversion (SCM software) (Makris, 2004).

The approach proposed in this dissertation still requires effort from the software producer to provide the information about tests, defects and requirements. Also, effort is required to analyse the information that is stored. However, the set of metrics is defined, and is based on information that is being created and used by the software development process (i.e. requirements, tests and defects). In addition to the metrics proposed here, additional information can be obtained from the SCM itself. Because this meta-data is stored with the source files, those wishing to analyse the source code to gather other metrics can do so by interrogating a single source of information: the SCM repository. Metrics other than those proposed here (such as number of lines of code) can easily be seen from the source code in the SCM. The set of metrics proposed here does not change over the course of the project, yet because they are based on the information in the SCM (which contains a lot more information that could be used for quality assessment) the set of metrics is extensible and may be added to, or subtracted from, over the course of a project. So the amount of effort required to define metrics based on the approach suggested here is small, but the developer, manager or sponsor (i.e. those wishing to gain insight into quality) is not bound by only these metrics and can store other information as required to attain insight into quality.

In all of the approaches discussed in Chapter 3, there is a need for manual measurement and analysis of the metrics. This presents a considerable barrier to the use of such approaches in light of the focus on cost and timelines in software projects. The approach discussed here describes an equation (see Equation 4.3) that can be used to indicate the quality of the software based on the metrics of requirements, tests and

defects. While the approach is not prescriptive about how to collect this information, this information is based on processes and activities that already form part of the software development lifecycle. A project must have requirements, should have tests to prove adherence to these requirements, and may or may not have defects showing non-conformance to the requirements.

Different development projects may use different development tools and the manner in which the meta-data is collected and stored is different for each. All that is required is that information about requirements, tests and defects are associated with source files as necessary. Requirements tracking, testing and defect management tools can be integrated with the SCM to manage this information. Also, integration with development environments could provide the developer with necessary prompts to provide the information as required. It is important to note that (as discussed in Section 2.3) the value of a process is evident only when that process is used, so it is ultimately up to the members in the project team to ensure that the information is stored and used correctly.

The approach discussed here attempts to address the major apparent shortcoming of the approaches discussed in Chapter 3: the effort required to define, collect and analyse metrics. It also attempts to provide an insight into the quality of the product by using metrics based on the product. This is done by defining a set of metrics, suggesting integration with existing development tools and using a quality equation. This may provide a "narrow" view of quality, in that the view of quality is based on a relatively small number of metrics, but the approach provides extensibility because the information that is stored in the SCM repository can be expanded beyond simply those metrics proposed by the approach.

4.6 Metaversion: a reference implementation

4.6.1 Collecting source file meta-data

As mentioned previously in this chapter, for each of the source files in a software project, the following information must be stored (as per the assumptions noted in Section 4.2):

- details of the tests that include or act upon the source file;
- details of the defects that the source file participate in — or contribute to; and
- details of the requirements that are claimed (and are proven using tests) to be met by the source file.

It should be borne in mind that there may need to be significant effort expended in order to determine which file gives rise to a defect, but this effort would have to be expended in the course of fixing the defect anyway.

Using Subversion, which has meta-data capabilities using *properties*, the required meta-data can be stored. The approach discussed here does not rely on Subversion being used as the Software Configuration Management system. The approach would work with any Software Configuration Management system that supports setting arbitrary metadata on individual files in the repository. Subversion was chosen for this reference implementation because of the author of this dissertation's familiarity with the software.

Subversion properties can have any name and are version-controlled along with the file itself. This means that over time, as the contents of the file change and requirements, tests and defects are associated with it, the values of the properties also change. The property names and values that are to be set against each version of a source file (as applicable) are listed below. Note that the identifiers referred to below are the test numbers, requirements numbers or defect numbers that identify each of the tests, requirements or defects in the project.

mv:tests-pass A list of identifiers for the tests that have run and are known to have passed for this version of the file.

mv:tests-total A list of identifiers for all of the tests that are associated with this version of the file.

mv:tests-fail A list of identifiers for the tests that have run and are known to have failed for this version of the file.

mv:defects-open A list of identifiers for the defects that have been reported against this version of the file and are as yet unresolved.

mv:defects-closed A list of identifiers for the defects that have been reported against this version of the file and are resolved.

mv:req-met A list of identifiers for requirements that have been proven to be fulfilled by this version of the file.

mv:req-unmet A list of identifiers for requirements that have been proven not to be fulfilled by this version of the file (e.g. after a defect has been raised against a previously met requirement).

mv:req-total A list of identifiers for requirements that this version of the file is supposed to fulfill.

The *mv:req-total* property would be set based on the requirements specification (hence the assumption in Section 4.2 that there should be an enumerated list of requirements — for easy reference). Based on these requirements, a set of tests would be created, and the *mv:tests-total* property would be set from these.

If testing were an automated process, the setting of the *mv:tests-pass* and *mv:tests-fail* properties could be done by the software tool performing the testing. If a defect management system was in use; when a defect is raised, it could set the value of the *mv:defects-open* property. Similarly, when a defect was resolved, the *mv:defects-closed* property could be set. The *mv:req-met* and *mv:req-total* properties could have their values automatically set by some requirements management tool, if one were in use. If there are details about which tests are related to which requirements, then details about met and unmet requirements can be inferred from passed and failed tests respectively. The properties *mv:req-unmet* and *mv:tests-fail* are used to store this information.

If no automated software is in use for these areas of the software project, the values could be set manually as part of the planning (for requirements and tests) and development and testing (for tests and defects).

Figure 4.1 depicts the collection of the properties from the various sources in the software project.

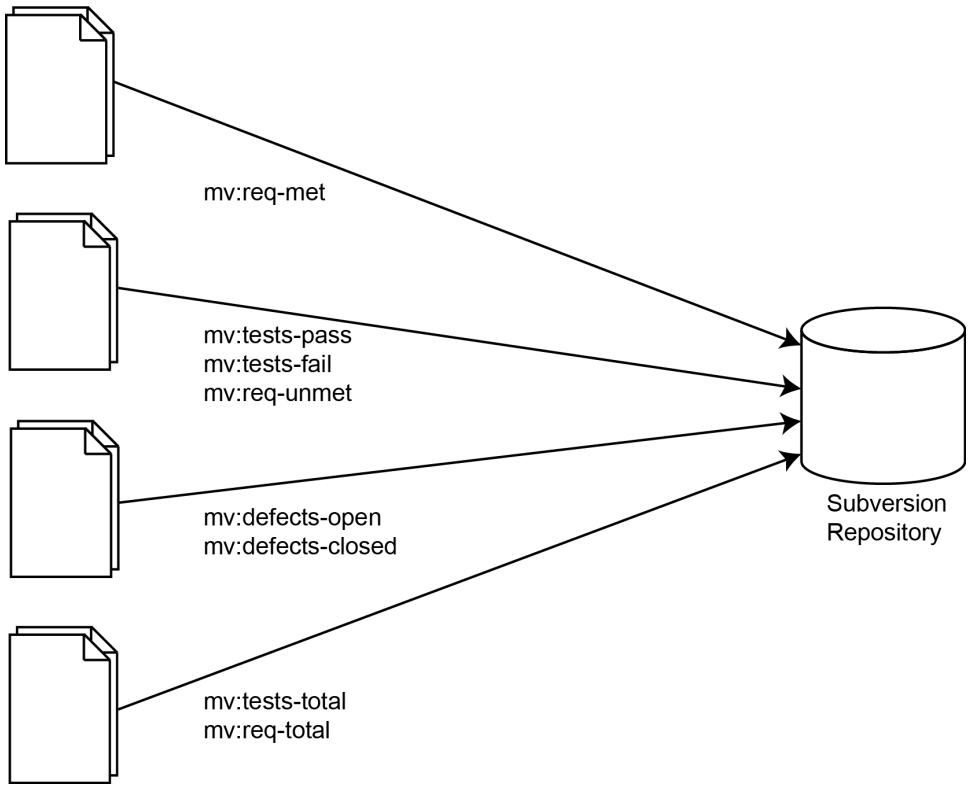

Figure 4.1: Gathering source file meta-data

4.6.2 Using source file meta-data

The value of storing the property meta-data is in the analysis thereof. Using the values of these properties and the quality equation described in Equation 4.3 (and any other metrics that can be derived from these properties), it is possible to get some idea of the quality of a particular source file, but perhaps of more value is the collective quality of the set of source files: the quality of the software product as a whole. This approach (as mentioned before) is not prescriptive regarding the mechanism for extracting the meta-data nor the tools to use to do it. This reference implementation is a suggestion of how this could be done.

One approach to using the source file meta-data would be to step through every file in the repository, read the values of the properties and build a report from that data. As the size of the repository grows, however, this approach would take longer to produce results because there are more source files to store meta-data for and hence more meta-data.

A more scalable approach would be to have the Subversion repository react when

a property is set on a file (and that property is one of the properties mentioned in Section 4.6.1) by storing the data in some intermediate location. The Subversion application provides this event-based mechanism through the use of *hooks*. These hooks allow custom applications to run when some activity occurs in the repository. The hook of interest to Metaversion is the *post-commit* hook. This hook is triggered when any file or change to a file is committed to the repository (including property changes). These hooks allow for an event-driven approach to the analysis of the meta-data as opposed to a polling approach. The data needed to perform analysis and reporting will be built up over time and the reporting application need only consult the intermediate store of data to build reports and not the entire Subversion repository.

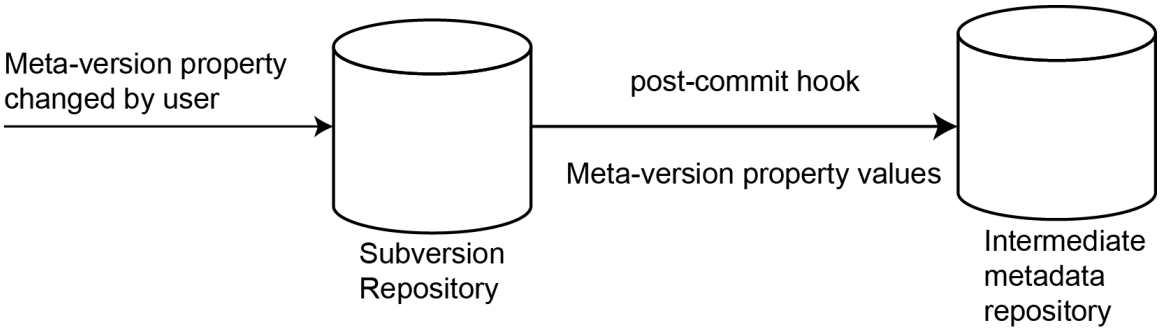

Figure 4.2: Reacting to post-commit hook

Figure 4.2 shows how the properties set on source files in the repository will be stored in an intermediate database. The post-commit hook will be configured to call an application. This application is called *MetaversionCommitHandler*, and is produced as part of this dissertation.

The meta-data stored with source files will be extracted using a reporting application. This reporting application will gather the meta-data from the repository and produce reports that will aid the development team, managers and sponsors of the software product to gain insight into the quality performance of the software. This reporting application is called *MetaversionReporter* and is produced as part of this dissertation.

4.6.3 Metaversion Data Model

Metaversion reflects the relationship between tests, defects and relationships as indicated in Figure 4.3. Each file can have zero or more tests, zero or more defects and zero or more requirements associated with it.

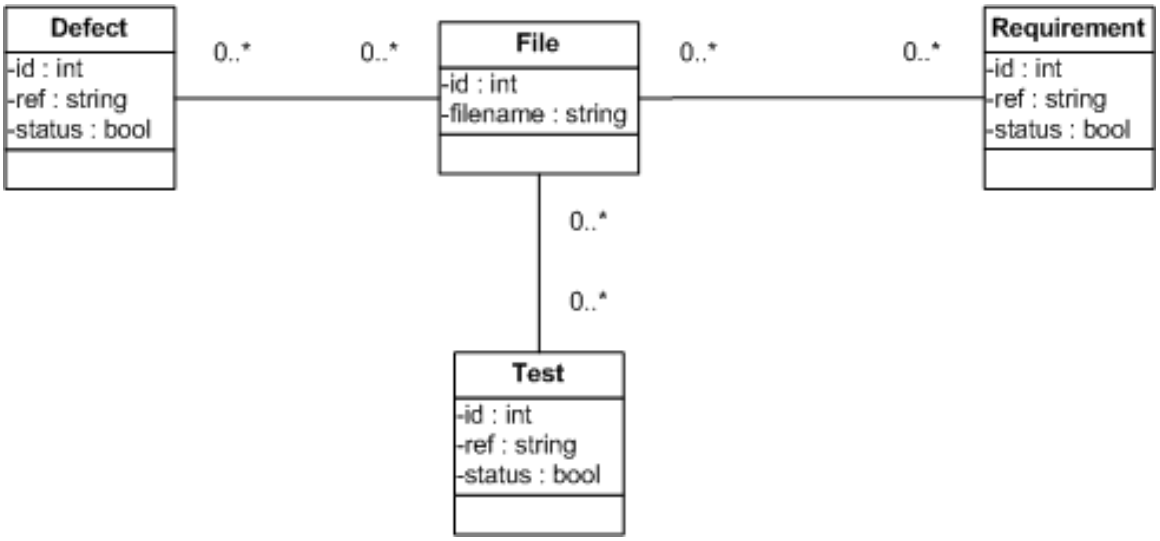

Figure 4.3: Metaversion data model

4.6.4 MetaversionCommitHandler

When a user (developer, manager, sponsor, etc.) sets a property on a file in the repository and commits the changes to the Subversion repository, the repository invokes the post-commit hook. The hook, in turn, invokes the *MetaversionCommitHandler* application. The *MetaversionCommitHandler* application calculates which properties have changed (in particular, the properties from the list in Section 4.6.1 for each file for which properties have been set. A MySQL (DuBois et al., 2008) database is updated with the details stored in the properties: requirements, tests and defects are associated with files, as appropriate. Figure 4.4 shows this process. *MetaversionCommitHandler* is used in the Case Study in Chapter 5.

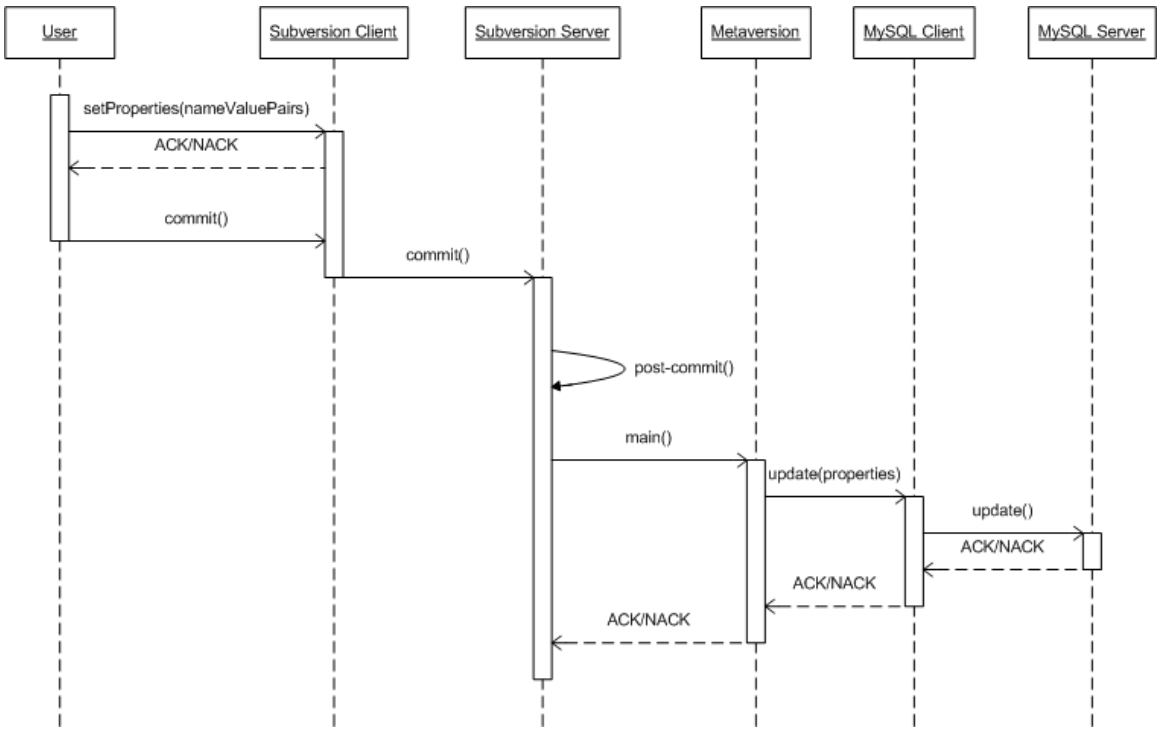

Figure 4.4: Metaversion commit process

4.6.5 MetaversionReporter

As mentioned, the value of the meta-data stored in the intermediate database is in the analysis thereof. The *MetaversionReporter* application is run by a member of the team developing the software (developer, manager, sponsor, etc.) in order to gain insight into the quality of the product. The *MetaversionReporter* application reads the information from the MySQL database that is populated by the *Metaversion-CommitHandler* application. This information is then used to calculate the value of Q (see Equation 4.3). This information is then displayed to the user. This process is depicted in Figure 4.5. *MetaversionReporter* is used in the Case Study in Chapter 5.

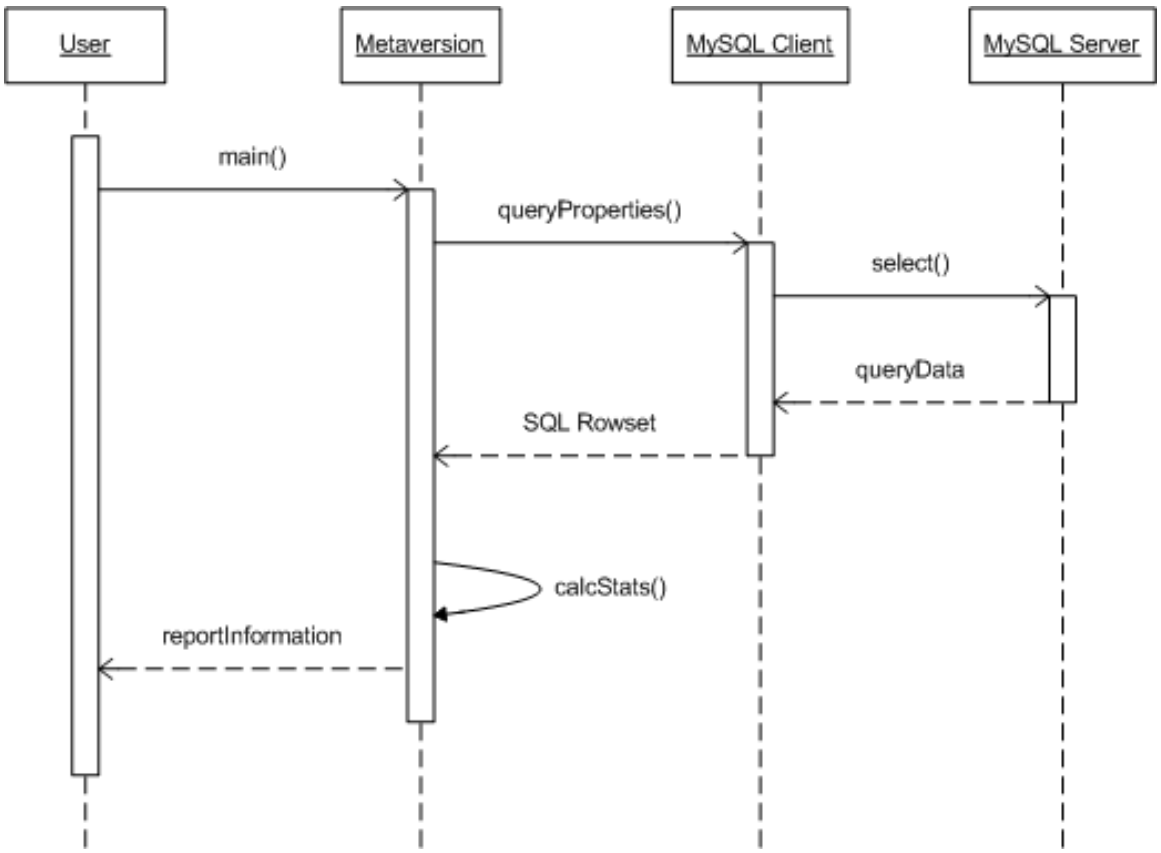

Figure 4.5: Metaversion reporting process

This chapter discussed a pragmatic approach to the analysis of quality in software projects. This approach suggests storing information about tests, defects and requirements for each source file in a Software Configuration Management repository. These metrics are used with the proposed equation for quality to provide an insight into the quality of the software produced. A reference implementation of the approach is provided, called Metaversion.

The next chapter will describe a case study of the approach described here.

Chapter 5

Case Study

This chapter presents a case study of the approach discussed in Chapter 4. The approach was used during the development of an Android[1] mobile application called *Timezonr*[2], for a customer named Richard Tasker, a personal friend of the author of this dissertation.

The Timezonr application was written using Java, with the Android SDK as a basis. The application was relatively small, consisting of 1364 lines of code, and was completed in 3 calendar months. The application consisted of 5 source code files and 7 configuration XML files. Of these files, only 3 source files had requirements set against them, with the other 2 being involved in auxiliary tasks not directly related to the requirements (such as presentation, string formatting and data classes). At the beginning of the project, there were 21 requirements and 21 tests and at the end of the project, there were 29 requirements and 29 tests. In total, there were 5 interim products and a final release version of the product. This particular project was chosen because it would be possible to complete it in a short amount of time, and with a small number of people involved. The team that was involved was the customer, a developer (the author of this dissertation) and a tester. The project that was chosen, although small, does represent a real-world project, given the large number of mobile applications available today (Whitney, 2011). More details of the Case Study project can be found in Appendix A.

The tests used were defined from the requirements and manually executed by the tester and customer. Testing was not performed in an automated manner, rather a human being performed tests by using the product after each release.

[1] http://www.android.com
[2] http://www.aravendi.com/timezonr.html

This chapter discusses the objectives of the case study, the methodology used during the case study and the data uncovered during the case study.

5.1 Objectives and Questions

The main objective of this case study is to provide some insight into the effectiveness of the approach proposed in Chapter 4 based on a real-world application of the approach. The case study aims to provide some understanding of what, if any, useful information is revealed by adopting the approach presented in this dissertation in a real-world project. Since there are suggestions that implementing software quality metrics are expensive and preclude small organisations (Staples et al., 2007), (Stelzer et al., 1996), this case study aims to take a small project with a small team and implement an approach to measuring quality. As mentioned in the analysis of Chapter 4, this approach attempts to address the major apparent shortcoming of the approaches discussed in Chapter 3: the effort required to define, collect and analyse metrics. It also attempts to provide an insight into the quality of the product by using metrics based on the product.

The specific question that this case study aims to investigate is:

- Does the implementation of the approach described in Chapter 4 yield results that resemble the user's perception of quality?

5.2 Methodology and Assumptions

Metaversion, the reference implementation discussed in Chapter 4, was used as a mechanism for gaining insight into the quality of software throughout the project. The project followed an agile software development life cycle (characterised by changing requirements, short release cycles, and test-driven development) that consisted of the steps enumerated below. These steps were repeated for each of the interim builds of the product.

1. Analysis and Design. In this phase, the requirements for the project were gathered. Consultation occurred in physical meetings and other communications between the customer and developer. The output of this phase was a list of requirements. The requirements were grouped by function and each requirement had a unique number associated with it. For each requirement, a test was created and numbered. The list of requirements was not static throughout the

project, but there was an initial set of requirements gathered so that work could commence.

2. Build. In this phase, source files were created and source code was written to fulfill the requirements identified in the Analysis and Design phase. Each file (as appropriate) was associated with a set of requirements and tests. During the build phase, the developer set the appropriate *Metaversion* properties for each file to reflect the association between files, requirements and tests.

3. Test. In this phase, the tests defined in the previous phase were executed by the tester and customer. Test results were noted and the appropriate *Metaversion* properties were set.

In the calculation of the value of Q (see Equation 4.3), a low value of 0.3 is chosen for α. This is to place higher weight on the value of T (the completeness confidence). A value of 0.3 was chosen because it provides T with a weight of slightly more than double the weight of R, implying that the importance of testing completion in the project was slightly more than twice as important as requirements completion.

The Build and Test phases were iterated to provide releases or interim products. There were occasions when requirements were changed, added to or removed from the requirements list. Once some portion of the code was written, a release was created and the team ran the tests against this interim product. Once the tests had been run, and all relevant *Metaversion* properties had been set on the source files, the *MetaversionReporter* application was run to extract a value for Q (see Equation 4.3).

At the same time, a questionnaire was answered by the product customer and tester (both tested the application). The questionnaire was developed as part of this dissertation. The respondents were both experienced software practitioners, one a product manager, the other an integration specialist. Given their extensive experience, it did not seem necessary to provide them with a notion of completeness, quality or confidence.

The questions are listed here.

1. **On a scale of 1 to 5, how complete is the product (where 1 indicates least complete and 5 indicates most complete)?** This question aims to assess the interviewee's perception of the level of completeness of the product. This level of completeness is what Equation 4.1 attempts to indicate by relating the requirements that the development team claim are fulfilled to the total number of requirements.

2. **On a scale of 1 to 5, how confident are you of the completeness you indicated in your previous answer (where 1 indicates not confident and 5 indicates very confident)?** This question aims to assess the interviewee's level of confidence in completeness, which gives some indication of whether the interviewee believes that other influences (than the requirements being fulfilled) may affect the completeness. This confidence is what Equation 4.2 aims to indicate. Equation 4.2 relates the number of tests that have passed with the total number of tests, and acts as a verification of the claim of completeness implied in Equation 4.1.

3. **On a scale of 1 to 5, how much of the functionality you wish to see is present (where 1 indicates none and 5 indicates all)?** This question aims to assess the degree to which the requirements of the users are successfully translated into the software as built. This question is similar to question 1, but question 1 is directed at the fulfilment of requirements while this question is aimed at the the interviewee's perception of whether the product serves their purposes. This question attempts to gain a view of quality in use as discussed in (International Organization for Standardization, 2011b).

4. **On a scale of 1 to 5, how much of the functionality you wish to see works correctly (where 1 indicates none and 5 indicates all)?** This question aims to assess the degree to which the the interviewee perceives that their needs (which may or may not be represented in the requirements) have been met correctly (i.e. where the need has been claimed to be fulfilled, it has been fulfilled correctly to the satisfaction of the interviewee). While question 3 aims to assess whether the functionality has been provided, this question aims to assess whether the functionality has been provided correctly.

5. **On a scale of 1 to 5, how much of the overall functionality you see in this release is tested by the tests you are executing (where 1 indicates very little and 5 indicates all)?** This question aims to assess the degree to which the interviewee perceives that the tests actually test the software.

6. **On a scale of 1 to 5, how would you rate the quality of the software (where 1 indicates low and 5 indicates high)?** This question aims to assess the level of quality that the interviewee perceives the software has.

After each build and test cycle, comparison of the answers to the questions mentioned above with the results of the *MetaversionReporter* application were performed to assess whether the quality perceived by the interviewee was similar to that indicated by the results calculated based on Equation 4.3. A similarity between what the interviewee perceives and what the *MetaversionReporter* application reports may suggest

that the approach discussed in Chapter 4 could be used to infer some information about the quality of the product.

The results of the questionnaire were taken as the average of the values of the responses of the customer and tester. The responses of the respondents were generally similar.

5.3 Threats to validity

The case study chosen was required to be representative of a small organisation that attempts to implement a software quality metrics model, since there are suggestions in the field that small organisations may be precluded from implementing quality metrics models (Staples et al., 2007), (Stelzer et al., 1996). However, because the team is small and the project is small (with a relatively small code base and number of requirements), it is not possible to state that the findings are representative of small projects or small organisations in general.

Also, because this case study only investigates the implementation of one quality metrics approach in one project, it is not possible to state that all projects would benefit from implementing the approach.

As mentioned in the analysis of Chapter 4, the set of metrics used by this approach would only provide a "narrow" view of quality. The metrics used (requirements, tests and defects) were chosen because in most projects, some effort would be expended in tracking these metrics anyway, hence they can be used without expending extra effort in calculation. However, the number of metrics is small in relation to other approaches to measuring quality using metrics, such as SQuaRE (International Organization for Standardization, 2011b). Thus, the metrics used in this case study may miss some elements of quality, and hence it is not possible to state that the quality calculated by the approach in Chapter 4, the Product Quality Indicator, can be applied to all projects, or that it provides a complete view of quality in general.

The survey executed alongside the case study had only two respondents. The survey is intended to achieve some notion of the quality of the product as perceived by the stakeholders of the project. It also uses an average function to calculate the responses of the respondents, though the pitfalls of using averages (discussed by Mordal-Manet et al. (2011) are mitigated somewhat because the responses were generally similar. Also, the fact that we cannot say that the difference between "almost none" and "none" is the same as the difference between "almost all" and "none" (as on a Likert scale) means that the use of averages may decrease the effectiveness of the survey. Because of the small sample size (of respondents) and the method of calculation of

responses, along with the fact that the survey results are naturally subjective (since they ask about the perception of the respondent), it is only possible to say that the survey provides a vague notion of the perceptions of users, and it is not possible to say that the perceptions are those of users in general.

As mentioned above, the author of this dissertation was also the developer in the case study. In order to remove bias, the person doing a case study should not be involved in it. This implies that there may be bias in the findings. For example, in a case study where the observer is not involved, it would be up to the development team to decide on the benefit of implementing a quality metrics model in general, and the one described in Chapter 4 in particular. Because of this bias, it is not possible to say that the approach would be applicable to all projects, since some projects may decide that the level of effort required to implement is too high.

The requirements for the case study did not vary much over the course of the project. There were 21 requirements at the beginning of the project and 29 at the end of the project. Because the requirements for each build were known to the survey respondents when they were assessing the quality of each build, this means that their perception of quality may be informed by the requirements, i.e. the survey may be imposing a view of quality that quality is related to fulfillment of requirements. This may limit the notion of quality in the survey results.

The respondents were both experienced software practitioners, one a product manager, the other an integration specialist. Since both are technically experienced, they may not be fully representative of the actual users of the application.

The purpose of this case study is to observe *in situ* the proposed approach being implemented in a real-world software project, and learn from the outcome, rather than claim generalised results. The size and nature of the project were chosen because of the level of time investment required (from the author, sponsor and tester), as well as the general expectation of the longevity of this type of research. As such, this case study does not aim to provide any evidence of generalisable results, but it does provide an initial investigation into the proposed approach, upon which future work (either by the author or others) might be based.

5.4 Specific data uncovered

The values for Q, Q_1, Q_2, Q_3 and Q_4 are the values for Equation 4.3, Equation 4.4, Equation 4.5, Equation 4.6 and Equation 4.7, respectively. The values shown here have been truncated to 3 decimal places. For all calculations below, a value of 0.3 is used for α. The results here are produced by the *MetaversionReporter* application

described in Section 4.6.5. The number of closed defects is a running total of the number of defects closed since the beginning of the project, and the number of open defects is the current number of outstanding defects. The number of requirements can change between builds, as requirements are added or removed. The value of D is the number of open defects. All counts are taken at the end of the testing of a build.

At the beginning of the development cycle, the *Metaversion* properties *mv:req-total* and *mv:tests-total* were set against each of the 3 source files. There were 21 requirements at the beginning of the project.

- Defects: $D = 0$, Open $= 0$, Closed $= 0$
- Tests: $T = \frac{T_{run} - D}{T_{total}} = \frac{0-0}{21} = 0$
- Requirements: $R = \frac{R_{met}}{R_{total}} = \frac{0}{21} = 0$
- $Q_0 = \alpha \times R + (1 - \alpha) \times T = 0.3 \times 0 + 0.7 \times 0 = 0 + 0 = 0$
- $Q_1 = \frac{R+T}{2} = \frac{0+0}{2} = 0$
- $Q_2 = \sqrt{R \times T} = \sqrt{0 \times 0} = 0$
- $Q_3 = \sqrt{R^2 + T^2} = \sqrt{0 + 0} = 0$
- $Q_4 = -\log_2\left(\frac{2^{-R}+2^{-T}}{2}\right) = -\log_2\left(\frac{2^{-0.000}+2^{-0.000}}{2}\right) = -\log_2(1.000) = 0.000$

5.4.1 Build 1

The first build of the project aimed to fulfill 5 requirements and set up a basic prototype application. One defect was found during testing.

- Defects: $D = 1$, Open $= 1$, Closed $= 0$
- Tests: $T = \frac{T_{run} - D}{T_{total}} = \frac{5-1}{21} = 0.190$
- Requirements: $R = \frac{R_{met}}{R_{total}} = \frac{5}{21} = 0.238$
- $Q_0 = \alpha \times R + (1 - \alpha) \times T = 0.3 \times 0.238 + 0.7 \times 0.190 = 0.071 + 0.133 = 0.205$
- $Q_1 = \frac{R+T}{2} = \frac{0.238+0.190}{2} = 0.214$
- $Q_2 = \sqrt{R \times T} = \sqrt{0.238 \times 0.190} = 0.213$

- $Q_3 = \sqrt{R^2 + T^2} = \sqrt{0.057 + 0.036} = 0.305$
- $Q_4 = -\log_2\left(\frac{2^{-R}+2^{-T}}{2}\right) = -\log_2\left(\frac{2^{-0.238}+2^{-0.190}}{2}\right) = 0.214$

The results of the questionnaire were as follows:

1. Completeness: 2
2. Confidence in completeness: 4
3. Completeness of specifications: 2
4. Fulfilment of specifications: 2
5. Test coverage: 2
6. Quality assessment: 3

5.4.2 Build 2

The second build of the project aimed to fix the defect found in build 1 and fulfill a further 17 requirements. No defects were found during testing (so the defect from build 1 was fixed and no new defects were uncovered). The second build had 4 more requirements than the first build.

- Defects: $D = 0$, Open $= 0$, Closed $= 1$
- Tests: $T = \frac{T_{run}-D}{T_{total}} = \frac{17-0}{25} = 0.680$
- Requirements: $R = \frac{R_{met}}{R_{total}} = \frac{17}{25} = 0.680$
- $Q_0 = \alpha \times R + (1-\alpha) \times T = 0.3 \times 0.680 + 0.7 \times 0.680 = 0.204 + 0.476 = 0.680$
- $Q_1 = \frac{R+T}{2} = \frac{0.680+0.680}{2} = 0.680$
- $Q_2 = \sqrt{R \times T} = \sqrt{0.680 \times 0.680} = 0.680$
- $Q_3 = \sqrt{R^2 + T^2} = \sqrt{0.462 + 0.462} = 0.962$
- $Q_4 = -\log_2\left(\frac{2^{-R}+2^{-T}}{2}\right) = -\log_2\left(\frac{2^{-0.680}+2^{-0.680}}{2}\right) = 0.680$

The results of the questionnaire were as follows:

1. Completeness: 3
2. Confidence in completeness: 4
3. Completeness of specifications: 3
4. Fulfilment of specifications: 3
5. Test coverage: 3
6. Quality assessment: 4

5.4.3 Build 3

The third build of the project aimed to fulfill 3 more requirements. One defect was found during testing. The number of requirements in build 3 was the same as the number of requirements in build 2, although 3 requirements were removed and 3 were added.

- Defects: $D = 1$, Open $= 1$, Closed $= 1$
- Tests: $T = \frac{T_{run} - D}{T_{total}} = \frac{20-1}{25} = 0.760$
- Requirements: $R = \frac{R_{met}}{R_{total}} = \frac{20}{25} = 0.800$
- $Q_0 = \alpha \times R + (1 - \alpha) \times T = 0.3 \times 0.800 + 0.7 \times 0.760 = 0.240 + 0.532 = 0.772$
- $Q_1 = \frac{R+T}{2} = \frac{0.800+0.760}{2} = 0.780$
- $Q_2 = \sqrt{R \times T} = \sqrt{0.800 \times 0.760} = 0.780$
- $Q_3 = \sqrt{R^2 + T^2} = \sqrt{0.640 + 0.578} = 1.103$
- $Q_4 = -\log_2\left(\frac{2^{-R}+2^{-T}}{2}\right) = -\log_2\left(\frac{2^{-0.800}+2^{-0.760}}{2}\right) = 0.780$

The results of the questionnaire were as follows:

1. Completeness: 4
2. Confidence in completeness: 4
3. Completeness of specifications: 4
4. Fulfilment of specifications: 4
5. Test coverage: 4
6. Quality assessment: 4

5.4.4 Build 4

The fourth build of the project aimed to fix the 1 open defect and fulfill a further 5 requirements. The defect that this build attempted to fix was marked as closed and then re-opened after testing, and a further 2 defects were found, leaving 3 open defects. The requirements remained unchanged between build 3 and build 4.

- Defects: $D = 3$, Open $= 3$, Closed $= 2$
- Tests: $T = \frac{T_{run} - D}{T_{total}} = \frac{25-3}{25} = 0.880$
- Requirements: $R = \frac{R_{met}}{R_{total}} = \frac{25}{25} = 1.000$
- $Q_0 = \alpha \times R + (1 - \alpha) \times T = 0.3 \times 1.000 + 0.7 \times 0.880 = 0.300 + 0.616 = 0.916$
- $Q_1 = \frac{R+T}{2} = \frac{1.000+0.880}{2} = 0.940$
- $Q_2 = \sqrt{R \times T} = \sqrt{1.000 \times 0.880} = 0.938$
- $Q_3 = \sqrt{R^2 + T^2} = \sqrt{1.000 + 0.774} = 1.332$
- $Q_4 = -\log_2\left(\frac{2^{-R}+2^{-T}}{2}\right) = -\log_2\left(\frac{2^{-1.000}+2^{-0.880}}{2}\right) = 0.939$

The results of the questionnaire were as follows:

1. Completeness: 4
2. Confidence in completeness: 5
3. Completeness of specifications: 5
4. Fulfilment of specifications: 4
5. Test coverage: 4
6. Quality assessment: 4

5.4.5 Build 5

The fifth build of the project aimed to fix the 3 open defects and fulfill a further 1 requirement. Testing revealed that the 3 open defects from build 4 were closed and 2 more were opened. Two new requirements were added in build 5.

- Defects: $D = 2$, Open $= 2$, Closed $= 5$
- Tests: $T = \frac{T_{run} - D}{T_{total}} = \frac{26-2}{27} = 0.889$
- Requirements: $R = \frac{R_{met}}{R_{total}} = \frac{26}{27} = 0.963$
- $Q_0 = \alpha \times R + (1 - \alpha) \times T = 0.3 \times 0.963 + 0.7 \times 0.889 = 0.289 + 0.622 = 0.911$
- $Q_1 = \frac{R+T}{2} = \frac{0.963+0.889}{2} = 0.926$
- $Q_2 = \sqrt{R \times T} = \sqrt{0.963 \times 0.889} = 0.925$
- $Q_3 = \sqrt{R^2 + T^2} = \sqrt{0.927 + 0.790} = 1.311$
- $Q_4 = -\log_2\left(\frac{2^{-R}+2^{-T}}{2}\right) = -\log_2\left(\frac{2^{-0.963}+2^{-0.889}}{2}\right) = 0.925$

The results of the questionnaire were as follows:

1. Completeness: 4
2. Confidence in completeness: 5
3. Completeness of specifications: 5
4. Fulfilment of specifications: 4
5. Test coverage: 5
6. Quality assessment: 5

5.4.6 Build 6

The final build aimed to fix all outstanding defects and fulfill all outstanding requirements. The open defects from build 5 were closed and no new defects were found. Two new requirements were added in build 6.

- Defects: $D = 0$, Open $= 0$, Closed $= 7$
- Tests: $T = \frac{T_{run} - D}{T_{total}} = \frac{29-0}{29} = 1.000$
- Requirements: $R = \frac{R_{met}}{R_{total}} = \frac{29}{29} = 1.000$
- $Q_0 = \alpha \times R + (1 - \alpha) \times T = 0.3 \times 1.000 + 0.7 \times 1.000 = 0.300 + 0.700 = 1.000$
- $Q_1 = \frac{R+T}{2} = \frac{1.000+1.000}{2} = 1.000$
- $Q_2 = \sqrt{R \times T} = \sqrt{1.000 \times 1.000} = 1.000$
- $Q_3 = \sqrt{R^2 + T^2} = \sqrt{1.000 + 1.000} = 1.414$
- $Q_4 = -\log_2\left(\frac{2^{-R}+2^{-T}}{2}\right) = -\log_2\left(\frac{2^{-1.000}+2^{-1.000}}{2}\right) = 1.000$

The results of the questionnaire were as follows:

1. Completeness: 5
2. Confidence in completeness: 5
3. Completeness of specifications: 5
4. Fulfilment of specifications: 5
5. Test coverage: 5
6. Quality assessment: 5

5.5 Results and Conclusions

5.5.1 Survey Results

Figures 5.1 through 5.6 show the responses of the survey with both respondents' answers and the averages of their answers. It can be seen that the answers of the respondents were in agreement for the majority of questions: a total 36 questions were asked across 6 builds, and the answers of the respondents were the same for 23 questions.

The results of questions 1, 2 and 6 are the subject of discussion later in this chapter. The results for the other questions are discussed here.

Question 3 aims to assess whether the respondent perceives that the build they're interacting with performs the functions that they wish to see. This gives an insight into whether the needs of the users were correctly translated into the requirements of the build. The results from question 3 are different for half of the questions. This might be attributable to the fact that this is an inherently subjective question, and the needs or desires of each of the respondents is different.

Question 4 aims to assess whether the functionality that has been provided to meet the user's needs has been implemented correctly. The results from question 4 are different for half of the questions. As with the responses to question 3, this could be the result of subjectivity and the respondents' needs or desires.

Question 5 aims to understand whether the respondents feel that the tests are appropriate to test all of the functionality implemented in a build. The results from question 5 are different for half of the questions. Since each of the requirements should have been associated with a test (the test proves that the requirement has been fulfilled), one could expect a response of 5 at all times. This is not the case which may indicate that the tests do not cover as much of the functionality in each build as the respondents would like. Respondent 2 has given increasing responses to question 5 as the project progresses, while Respondent 1 has given the same (almost all) response for all but the final build. The reason for this could be that the the respondents know about all of the requirements, and perceive that the number of tests does not match the total number of requirements without considering that the tests are intended to test only those requirements that are claimed to be met.

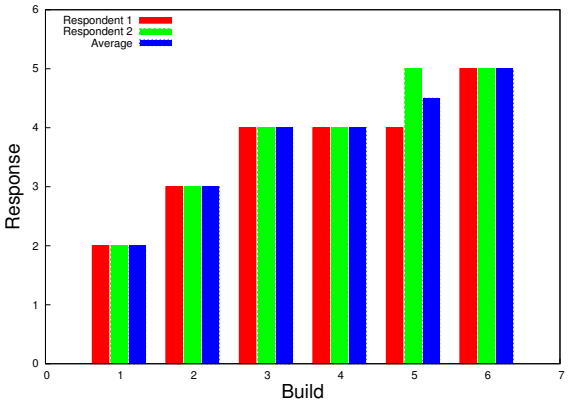

Figure 5.1: Question 1 responses

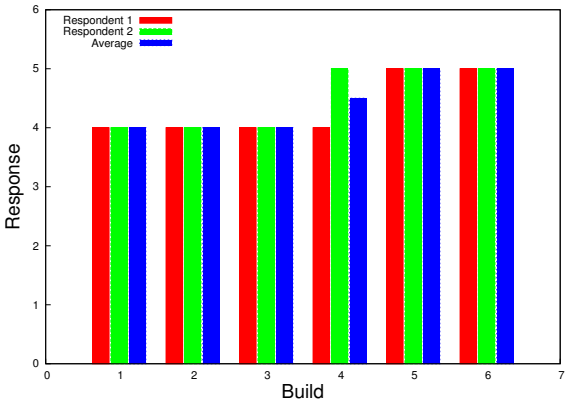

Figure 5.2: Question 2 responses

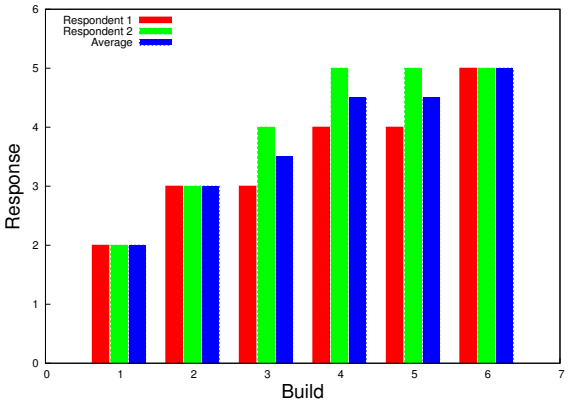

Figure 5.3: Question 3 responses

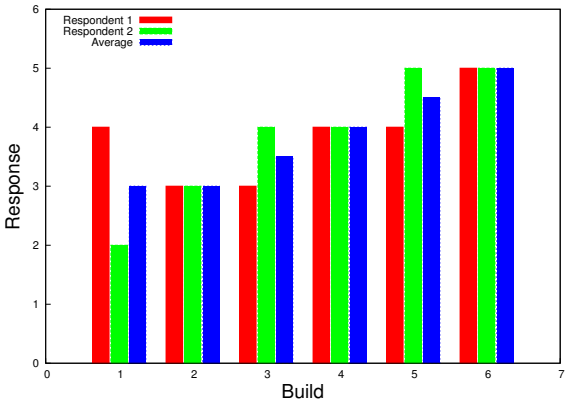

Figure 5.4: Question 4 responses

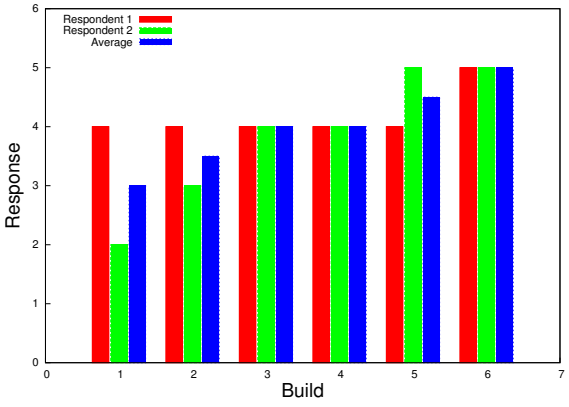

Figure 5.5: Question 5 responses

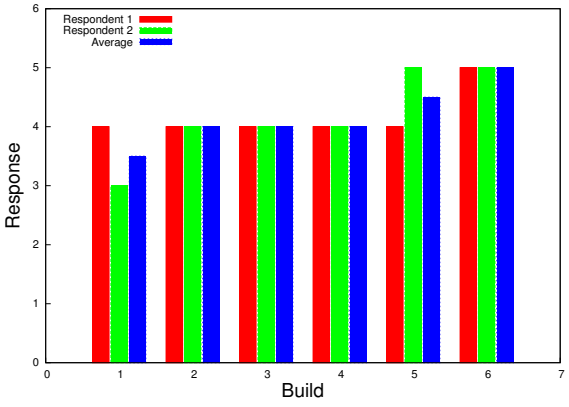

Figure 5.6: Question 6 responses

5.5.2 Quality

Figure 5.7 shows how the quality as perceived by the customer and tester increased as the build iterations were tested. The values plotted in this graph are the values of Q (and the alternative equations for Q discussed in Chapter 4.4) against the average of the values given as responses to question 6 of the questionnaire by the customer and tester. The values of Q, Q_1, Q_2 and Q_4 are very close to each other, with Q, Q_1 and Q_4 being mostly indiscernible from each other on the graph.

The perceived quality increases as the product is developed. This implies that the perception of the customer and tester is that as the project is being developed, the product produced gets closer and closer to what they want. The calculated quality also increases as the product is being developed. This is because, as the product is developed, more of the requirements are being met, and more of the tests indicating adherence to requirements are passing.

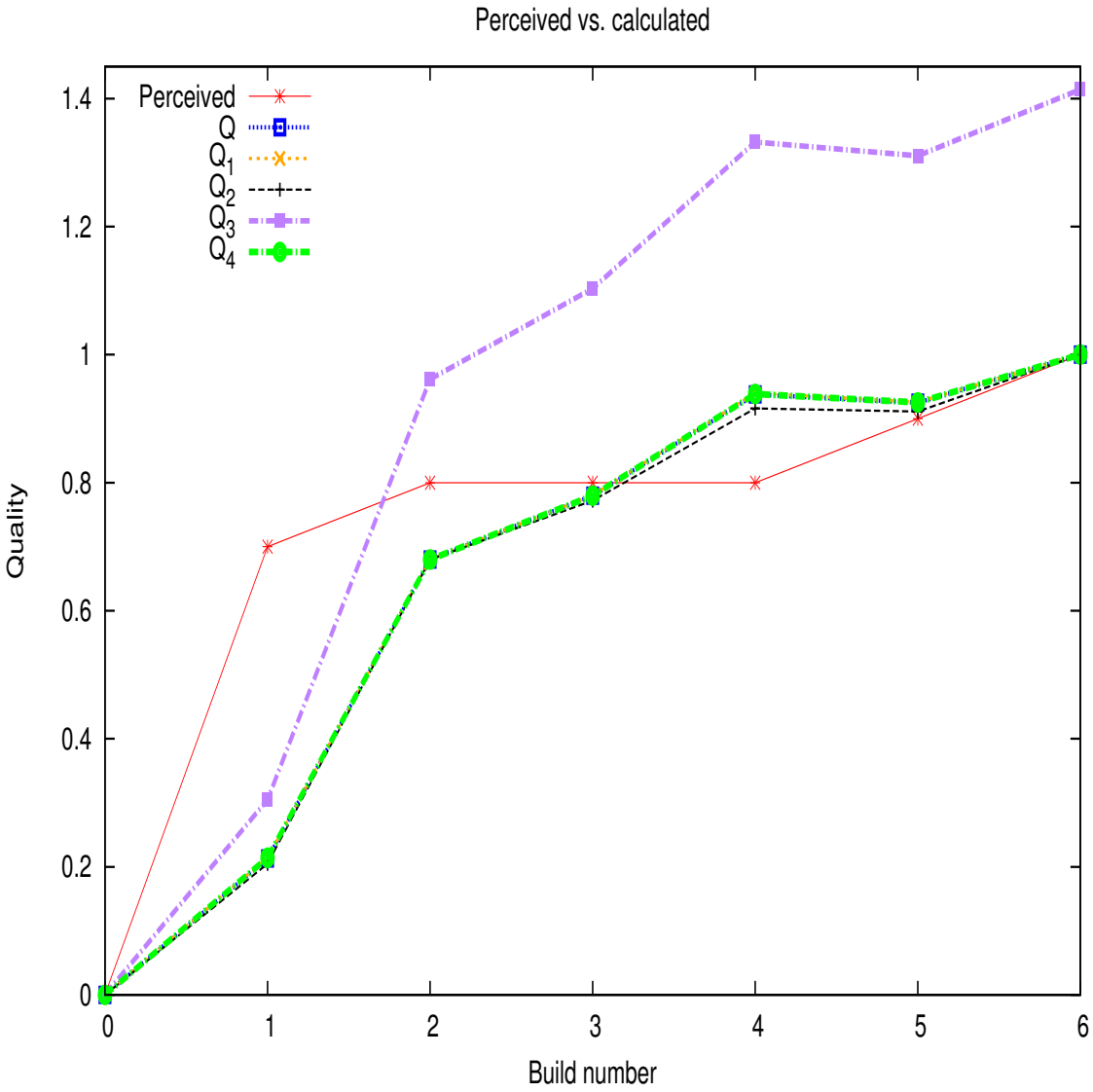

Figure 5.7: Quality: perceived vs. calculated

5.5.3 Completeness

Figure 5.8 shows how the completeness of the product is perceived by the customer and tester and the calculated values of completeness. The values plotted in this graph are the values of R against the average of the values given as responses to question 1 of the questionnaire by the customer and the tester. In general, both perceived and calculated completeness increase over the course of the project, and this indicates that the customer and tester believe that the project is progressing towards a state they believe is complete, and that the number of requirements (which are known to the testers) being met is increasing over the course of the project. At build 4 and 5, the perceived completeness was less than the calculated completeness. This could well be because of the combination of the customer's and tester's understanding of the term *completeness* and the discreet interval from which they had to choose values for completeness.

The calculated and perceived values are only equal once other than the start and end of the project. At build 4, the calculated completeness is 1.0, which implies that all requirements are claimed to have been met. The perception of the respondents, however, is that the project is not complete. Since the respondents know the requirements, this difference could be attributable to the presence of defects. The respondents know that the software has defects, and they may consider that this means the software is incomplete.

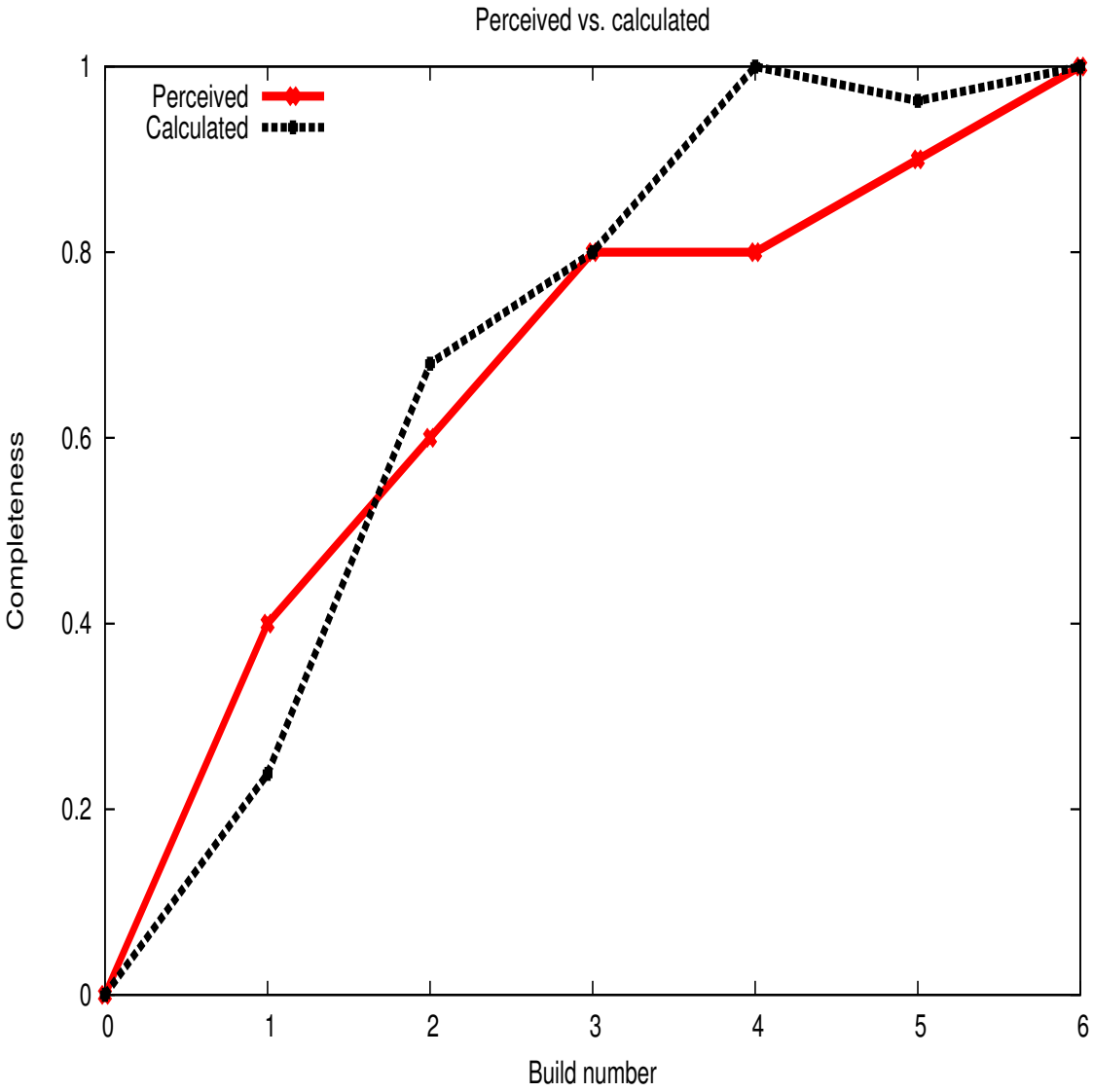

Figure 5.8: Completeness: perceived vs. calculated

5.5.4 Completeness confidence

Figure 5.9 indicates that the confidence in completeness increases as the project progresses. The values plotted in this graph are the values of T against the average of the values given as responses to question 2 of the questionnaire by the customer and the tester. The customer and tester perceive that they are more sure of their assessment of completion as the project progresses. This could be due to familiarity with the product, and an increase in understanding of how the application works. In addition to this, this measure indicates that the customer and tester feel they have enough of a view of the completeness of the product to be sure of it. For the calculated confidence in completion, this indicates that a higher proportion of the tests that are being run are passing, which lends some degree of confidence to the completeness of the product.

The values of the completeness confidence from the respondents are all high. This could be attributed to two things. Firstly, the responses only have discrete values, and so the respondent wouldn't be able to give an answer inbetween the values. Thus, they may, for example, choose 4 out of 5, when they actually wish to give a response of 3.5. Secondly, the respondents know the requirements of the build when they are testing, so they can actually prove their answer to question 1. Since all of the responses are not 5 out of 5, this could imply they did not do this, or it could imply that there is some other factor influencing their understanding of the question.

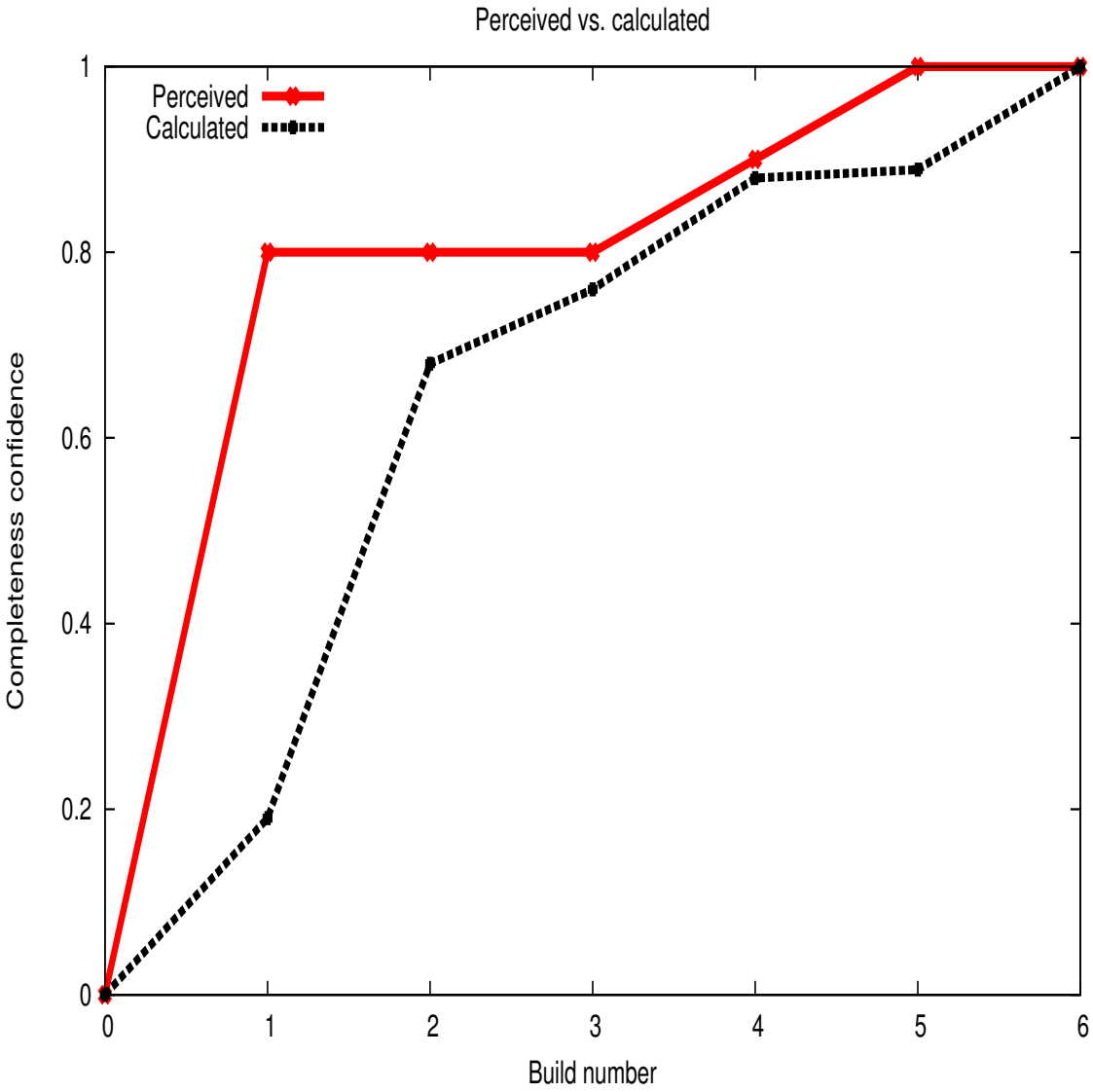

Figure 5.9: Completeness confidence: perceived vs. calculated

5.5.5 Conclusion

From the data revealed during this case study, there appears to be an increase in perceived and calculated quality, completeness and completeness confidence as the product is being developed. This increase, from the perspective of the customer and tester, could mean that as the project progressed, they were more confident that the software being produced met the requirements they specified, and that the quality of the product increased. From the perspective of the *MetaversionReporter*, the increase is due to more requirements being fulfilled and more tests passing.

Figure 5.7 shows that there is very little difference between the alternative methods for calculating Q discussed in Chapter 4.4, with the exception of Equation 4.6 represented in the graph by Q_3. This is expected, because the data do not represent extreme cases, for example where R is 1 and T is 0.

In this instance, it does seem as though as the calculated quality increases, so does the perceived quality. This provides *prima facie* evidence to support the assertion that *Metaversion* may be used to imply some information about quality, and that the information it provided was generally in line with what the customer and tester perceived. It should be borne in mind, though, that this case study is not representative enough to draw any conclusions regarding a general correlation between perceived and calculated quality. The case study was about a small project, with relatively few requirements, files, tests, line of code and team members. In order to test such a correlation, a number of larger projects would need to use *Metaversion*. The next chapter will discuss possible future research.

This chapter presented a case study using the *Metaversion* reference implementation discussed in the previous chapter. A software project was undertaken and the customer and tester's perceptions of quality, completion and their confidence in the completion of the product were recorded after each interim release of the product. The customer and the tester also executed a set of tests after each interim release and the results of these tests were used to set the values of the *Metaversion* properties and calculate a value for these metrics based on the equations proposed by the approach in the previous chapter.
The case study found a similar progression of the customer and tester's perception of quality and the value calculated by the *Metaversion* approach.
The next chapter will revise the material covered in this dissertation and provide some ideas for future work on the approach discussed.

Chapter 6

Conclusions and future work

The three main areas of interest to software quality that this dissertation covered were process, quality and metrics.

A process is the systematic application of a set of tasks or sub-tasks with the aim of converting an input to an output. A process directs the activities of a software engineering project and ensures that as far as possible from the project's inception, until the project's completion, the set of tasks that needs to occur are known. A benefit of a process is that it is repeatable.

Many software engineering projects have been completed since the inception of the computer. The experience of the people working on those software projects has given rise to a number of software engineering process models that software producers use to produce software products. A software engineering process, in some sense, is the distillation from experience of repeated tasks over a number of similar endeavours. The fact that many process models exist (a sample of which were discussed in Chapter 2.4) is testament to the importance of processes in software projects.

As the needs of customers and users change and evolve, software must evolve with it. As computing power increases and hardware architectures change, software must change too. Software producers (like manufacturers of other types of products) are driven to achieve maximum efficiency with their given resources. For this reason, the process used to create software must evolve over time. The software producer must take care to ensure that the processes that are in place in the production of software are managed and change when they need to. The quality standards discussed in Chapter 2 advocate the need for process (process being important to ensure consistent-quality outputs), but also the need for managing process (process management being necessary to ensure consistent and increasing quality outputs).

It is natural for any profit-driven concern to want to ensure that the products they

produce are high quality. Quality is a differentiating factor in the eyes of consumers of goods and services, and software quality is no less important. If a producer of software wishes for consumers to use their software and not the software of its competitors, then quality may be a reason that they would or would not. A question, then, is how producers of software ensure that they produce high quality software. There are many factors involved in creating software, and to say that implementing a process model would ensure quality would be to under-represent the complexities of all of the factors involved in software production. However, implementing a process model does guide the activities of software production and ensure that it is understood how the resources involved with producing software should be involved. A process model allows the complexities of creating software to be better managed by making it more clear what resources are doing which tasks at what time. If a process is in place, it may be easier to identify resources whose tasks may be performed better or more effectively, with a view to increasing the quality of the outputs.

Measurement and metrics play an important role for software producers in understanding the quality of their products. If the software producer understands that a particular resource is not performing as efficiently as it could, actions could be taken to remedy that. Having a set of metrics and measurements guides decisions because it allows decisions to be made based on facts. The same can be said for the process model used to produce software — if a task or set of tasks is taking longer than it should, actions could be taken to remedy that. The fact-based decision making capability afforded by using metrics and measurements can be used to increase the quality of outputs by ensuring that the resources and process models used are performing as effectively as they could be.

Process and metrics are two important parts of the effort involved in producing software that work together to help software producers create software that has consistent or increasing quality. Having a process and using metrics in themselves will not guarantee high quality outputs. The software producer must undertake to manage the process and collect and analyze metrics in order to constantly improve the process used to produce software. Chapter 3 discussed approaches that have been proposed by others in the field to help software producers understand how they can use process and metrics to aid them in achieving quality outputs.

Requirements should be a part of any software engineering project — they state what the customer and users wish the software to do. The requirements could be used as an indication of whether the software that is produced by the software producer is what the customer asked for. Tests are a useful part of any software engineering project — they let the software producer know which parts of the stated functional requirements have been fulfilled. When a test is written, it is written based on a stated functional requirement and the failure of a test signals that a requirement is

not being met. Defects, then, are the results of tests failing — they codify information about the failure of a test, indicating which requirement is not being met and guiding the software producer in rectifying the cause of the requirement not being met.

Software Configuration Management should be part of any software engineering project. The outputs of a project, the source files, the documentation and the tests can be stored in a Software Configuration Management system. Doing so has many benefits, not least of which is the focus on change management. These outputs of the project would naturally change over time — the requirements may change and the source files may change, among others. One reason to manage the inevitable change to these project artifacts is that the software producer may be able to understand more easily the cause of test failures. For example when a test fails because of a particular source file change, it may be easier to isolate the change that caused the issue.

In Chapter 4, an approach was discussed that built on those approaches discussed in Chapter 3. The approach used the requirements, tests and defects associated with a software product to deduce metrics that attempted to give insight into the quality of the software product after each interim release. The approach proposed using a Software Configuration Management system to store information about these metrics. A Software Configuration Management system, along with the requirements, tests and defects are usually a part of the software engineering project, and this means that the approach addressed (in some way) the barrier to implementing metrics and measurements that the approaches discussed in Chapter 3 raised.

In Chapter 5, a case study was discussed that used an implementation of the approach discussed in Chapter 4. The case study found that as the quality as calculated by applying the approach discussed in Chapter 4 increased, the quality as reported by the testers increased.

While the evidence uncovered by the case study does suggest that the approach proposed in Chapter 4 could be used to gain some insight into the quality of a software product, this evidence needs to be more strongly correlated with the reported quality in order to claim that the quality metrics reported by the approach proposed in Chapter 4 are actually an indicator of quality. With just one sample project and a small number of testers, it could be claimed that the fact that the quality reported by testers and that measured using the approach proposed in Chapter 4 is circumstantial and not repeatable. In order to test whether this correlation really exists, there needs to be a much larger number of testers expressing their opinions about quality, and of projects against which the proposed approach is tested. This may be an interesting extension of this research: to test the correlation between the results of the application of the proposed approach and the quality reported by testers. The case study also relies on human beings to describe their opinion about quality, and this means that the "perceived" quality is susceptible to human factors and biases based on factors

not directly related to the software product at all.

Another possible future area of research related to this approach is to investigate other metrics that could be gathered and combined to make the results calculated by the application of the approach proposed in Chapter 4 more closely correlate to the quality reported by testers. One such metric may be the number of developers involved in developing a particular source file. It may be useful to understand if the number of developers that are involved in delivering a requirement influences the quality of the product.

As mentioned above, the approach proposed in Chapter 4 does aim to lower the barrier to implementing metrics and measurements in software projects by making the collection and use of metrics part of the software development process. However, there is still some effort required to associate files, requirements, and tests with each other, and perhaps some future work could address decreasing this effort. The approach would be more extensive if there was a way to define arbitrary metrics, and ways of calculating quality indicators based on those, rather than using the static quality equation proposed with the inputs of tests, requirements and defects. Also, the extensibility of the approach would be increased by providing some way of interfacing with requirements management software and test and defect management software so that software producers are free to use systems that aid in easier management of requirements, tests and defects.

This dissertation covered software quality and proposed an approach to gaining some insight into the quality of a software product based on metrics that could easily be calculated using measures and other metrics that are inherently present in most software development processes. The outcome of the proposed approach, based on a case study performed on a software project, did yield some information that could indicate the level of quality a product exhibits.

Appendix A

Case Study materials

A.1 Case Study project information

The Case Study project involved creating an Android[1] application to easily display time zone information. This application is called *TimeZonr*[2]. The source code was organized into an Eclipse project[3], and the product of the project was an application package for use on an Android device.

A.1.1 List of requirements

As the case study project progressed, some requirements were added and some were removed. The list of requirements for the first release and the last release are reproduced here.

Table A.1: List of requirements for first release

Requirement number	Description
1	The application shows a table of timezones. Each column represents a timezone and each row represents an hour of the day in each configured timezone.
2	The default date for the application is the date at time of running the application.

[1]See http://www.android.com
[2]http://www.aravendi.com/timezonr.html
[3]See http://www.eclipse.org

Table A.1: List of requirements for first release

Requirement number	**Description**
3	The user can change the date for which the table shows information.
4	The application must respect Daylight Savings Time across different countries.
5	The current timezone (i.e. the timezone the user is in) is shown as the first column of the table.
6	The user can add multiple timezones, defined by
6.1	— their offset from UTC
6.2	— a short display name
6.3	— green hours: a set of working hours where meetings are acceptable in the timezone
6.4	— yellow hours: a set of hours before/after green hours where meetings are somewhat acceptable in the timezone
6.5	— red hours: a set of hours before/after yellow hours where meetings are not acceptable in the timezone
6.6	— Defaults for hours are:
	— Green: 8am – 5pm
	— Yellow: 7am – 8am, 5pm – 10pm
	— Red: 10pm – 7am
7	Times are shown with background colours of green, red or yellow.
8	User-added timezones are appended as columns to the right of existing timezones, with the table scrolling up/down and left/right as the table grows.
9	Configured timezones must be saved and displayed when the application is restarted.
10	The next available hour (current hour rounded up) is highlighted in the middle of the table of times.
11	Times that are in bold are the following day.
12	The user can select an option to show the next available meeting time (i.e. a time where no timezones are red). This will highlight the appropriate row in the table.

Table A.1: List of requirements for first release

Requirement number	Description
13	The user can select an option to show the next best meeting time (i.e. a time where there is at least one green time outside of the current time zone and no red times). This will highlight the appropriate row in the table.
14	Where a timezone has a display name (shown in the first row of the table) of more than 8 characters, ellipses must be appended when displaying in the table of hours.
15	If the user selects to show the next available or best meeting time and non exists, a message must be displayed to the user

Table A.2: List of requirements for last release

Requirement number	Description
1	The application shows a table of timezones. Each column represents a timezone and each row represents an hour of the day in each configured timezone.
2	The default date for the application is the date at time of running the application.
3	The user can change the date for which the table shows information.
4	The application must respect Daylight Savings Time across different countries.
5	The current timezone (i.e. the timezone the user is in) is shown as the first column of the table.
6	The user can add multiple timezones, defined by
6.1	— their Java timezone equivalent
6.2	— a short display name
6.3	— green hours: a set of working hours where meetings are acceptable in the timezone
6.4	— yellow hours: a set of hours before/after green hours where meetings are somewhat acceptable in the timezone
6.5	— red hours: a set of hours before/after yellow hours where meetings are not acceptable in the timezone

Table A.2: List of requirements for last release

Requirement number	**Description**
6.6	— Defaults for hours are:
	— Green: 8am – 5pm
	— Yellow: 7am – 8am, 5pm – 10pm
	— Red: 10pm – 7am
6.7	— An offset from UTC must be specified in format HH:mm, defaulting to 00:00
6.8	— When the add timezone wizard shows, it should revert to default values
7	Times are shown with background colours of green, red or yellow.
8	User-added timezones are appended as columns to the right of existing timezones, with the table scrolling up/down and left/right as the table grows.
9	Configured timezones must be saved and displayed when the application is restarted.
10	The next available hour (current hour rounded up) is highlighted in the middle of the table of times.
11	Times that are in bold are the following day.
12	The user can select an option to show the next available meeting time (i.e. a time where no timezones are red). This will highlight the appropriate row in the table.
13	The user can select an option to show the next best meeting time (i.e. a time where there is at least one green time outside of the current time zone and no red times). This will highlight the appropriate row in the table.
14	Where a timezone has a display name (shown in the first row of the table) of more than 8 characters, ellipses must be appended when displaying in the table of hours.
15	If the user selects to show the next available or best meeting time and non exists, a message must be displayed to the user
16	The first (header) row in the table shows the name of the timezone and its offset from UTC
17	Alternating rows of the table must be shaded grey and white respectively

Table A.2: List of requirements for last release

Requirement number	Description
18	It must be easy to add timezones to the list of viewed timezones, searching for cities by typing in a few letters of the city's name
19	The user has the choice to delete all of the configured timezones
20	The user has the choice to delete a single timezone by long-clicking the column header
21	The user must be able to select the current time (i.e. highlight the next hour regardless of availability)

A.1.2 Files in the project

The project consists numerous files, some of which had requirements associated with them. The table below shows the directories and files used in the project.

The **AndroidManifest.xml** file provides some for the application, including application name and the path to the application icon.

This directory provides high, medium and low density versions of the icons and graphics used in the application. It also provides layout details of the screens in the application, menu configurations and externalised string values.

```
./res/drawable-hdpi
./res/drawable-hdpi/ic_menu_add.png
./res/drawable-hdpi/ic_menu_agenda.png
./res/drawable-hdpi/ic_menu_delete.png
./res/drawable-hdpi/ic_menu_home.png
./res/drawable-hdpi/ic_menu_search.png
./res/drawable-hdpi/ic_menu_today.png
./res/drawable-hdpi/icon.png
./res/drawable-hdpi/timezonr_launcher_icon.png
./res/drawable-ldpi
./res/drawable-ldpi/ic_menu_add.png
./res/drawable-ldpi/ic_menu_agenda.png
./res/drawable-ldpi/ic_menu_delete.png
./res/drawable-ldpi/ic_menu_home.png
./res/drawable-ldpi/ic_menu_search.png
./res/drawable-ldpi/ic_menu_today.png
```

```
./res/drawable-ldpi/icon.png
./res/drawable-ldpi/timezonr_launcher_icon.png
./res/drawable-mdpi
./res/drawable-mdpi/ic_menu_add.png
./res/drawable-mdpi/ic_menu_agenda.png
./res/drawable-mdpi/ic_menu_delete.png
./res/drawable-mdpi/ic_menu_home.png
./res/drawable-mdpi/ic_menu_search.png
./res/drawable-mdpi/ic_menu_today.png
./res/drawable-mdpi/icon.png
./res/drawable-mdpi/timezonr_launcher_icon.png
./res/layout
./res/layout/add_tz_dialog.xml
./res/layout/custom_title.xml
./res/layout/main.xml
./res/layout/tz_record_row.xml
./res/menu
./res/menu/options_menu.xml
./res/values
./res/values/strings.xml

This directory contains the Java source code that was written for the project.
./src
./src/com
./src/com/danielacton
./src/com/danielacton/android
./src/com/danielacton/android/timezonr
./src/com/danielacton/android/timezonr/AddTimeZoneDialog.java
./src/com/danielacton/android/timezonr/DateSetListener.java
./src/com/danielacton/android/timezonr/TimezoneData.java
./src/com/danielacton/android/timezonr/TimeZonr.java
./src/com/danielacton/android/timezonr/Utils.java
```

A.1.3 Project progression

At the beginning of each release cycle, requirements were set against the files (being the requirements that those files were intended to fulfill). As the project progressed through the releases, requirements were met, some tests passed, some tests failed, and defects were raised and fixed. Tests in this project were not automated, but were run by humans interacting with the software. A set of tests was created, one test for each

requirement, to test that the functionality required was present. For ease of reference, the requirement numbers and test numbers were the same (i.e. the identifier for a test was the identifier of the requirement it was matched with). Similarly, the defect numbers matched the test numbers.

For the sake of brevity, the details shown below are shown for only the first and last release of the software.

A.1.3.1 Assignment of requirements to files

The table below shows the assignment of requirements to files at the beginning of the last iteration of the project.

Table A.3: Assignment of requirements to files

Requirement number	File
1	src/com/danielacton/android/timezonr/TimeZonr.java
2	src/com/danielacton/android/timezonr/TimeZonr.java
3	src/com/danielacton/android/timezonr/DateSetListener.java
4	src/com/danielacton/android/timezonr/TimeZonr.java
5	src/com/danielacton/android/timezonr/TimeZonr.java
6	src/com/danielacton/android/timezonr/AddTimeZoneDialog.java
6.1	src/com/danielacton/android/timezonr/AddTimeZoneDialog.java
6.2	src/com/danielacton/android/timezonr/AddTimeZoneDialog.java
6.3	src/com/danielacton/android/timezonr/AddTimeZoneDialog.java
6.4	src/com/danielacton/android/timezonr/AddTimeZoneDialog.java
6.5	src/com/danielacton/android/timezonr/AddTimeZoneDialog.java
6.6	src/com/danielacton/android/timezonr/AddTimeZoneDialog.java
6.7	src/com/danielacton/android/timezonr/AddTimeZoneDialog.java
6.8	src/com/danielacton/android/timezonr/AddTimeZoneDialog.java
7	src/com/danielacton/android/timezonr/TimeZonr.java
8	src/com/danielacton/android/timezonr/TimeZonr.java
9	src/com/danielacton/android/timezonr/TimeZonr.java
10	src/com/danielacton/android/timezonr/TimeZonr.java
11	src/com/danielacton/android/timezonr/TimeZonr.java
12	src/com/danielacton/android/timezonr/TimeZonr.java
13	src/com/danielacton/android/timezonr/TimeZonr.java

Table A.3: Assignment of requirements to files

Requirement number	File
14	src/com/danielacton/android/timezonr/TimeZonr.java
15	src/com/danielacton/android/timezonr/TimeZonr.java
16	src/com/danielacton/android/timezonr/TimeZonr.java
17	src/com/danielacton/android/timezonr/TimeZonr.java
18	src/com/danielacton/android/timezonr/AddTimeZoneDialog.java
19	src/com/danielacton/android/timezonr/TimeZonr.java
20	src/com/danielacton/android/timezonr/TimeZonr.java
21	src/com/danielacton/android/timezonr/TimeZonr.java

A.1.3.2 Requirements met, tests and defects

The requirements met, tests passed and failed, and defects raised and closed properties were set on the corresponding files as the project progressed. The lists below show the allocation of these properties to each of the files after the testing of the first and last build.

Table A.4: Metaversion properties per file after testing of first release

File	Requirements Met	Tests Passed	Tests Failed	Defects Opened
TimeZonr.java	1, 2, 5, 10, 14	1, 2, 5, 10	14	14
DateSetListener.java	3	3	-	-
AddTimeZoneDialog.java	-	-	-	-

Table A.5: Metaversion properties per file after testing of last release

File	Requirements Met	Tests Passed	Tests Failed	Defects Opened
TimeZonr.java	1, 2, 5, 10, 14, 4, 8, 11, 16, 19, 9, 17, 7, 12, 15, 13, 20, 21	1, 2, 5, 10, 14, 4, 8, 11, 16, 19, 9, 17, 7, 12, 15, 13, 20, 21	-	-
DateSetListener.java	3	3	-	-
AddTimeZoneDialog.java	6, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 18, 6.7, 6.8	6, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 18, 6.7, 6.8	-	-

Bibliography

Abran, A., Moore, J. W., Bourque, P., Dupuis, R. (Eds.), 2004. Guide to the Software Engineering Body of Knowledge. IEEE Computer Society.

Al-Qutaish, R. E., 2009. An investigation of the weaknesses of the iso 9126 international standard. In: Proceedings of Second International Conference on Computer and Electrical Engineering ICCEE'09. Vol. 1. IEEE, pp. 275–279.

Azume, M., Apr. 2001. Square: The next generation of the iso/iec 9126 and 14598 international standards series on software product quality. In: Proceedings of European Software Control and Metrics (ESCOM).

Barkmann, H., Lincke, R., Löwe, W., 2009. Quantitative Evaluation of Software Quality Metrics in Open-Source Projects. In: International Conference on Advanced Information Networking and Applications Workshops. pp. 1067–1072.

Beck, K., Andres, C., 2005. Extreme Programming Explained, 2nd Edition. Addison Wesley.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D., 2001a. Manifesto for Agile Software Development.
URL `http://www.agilemanifesto.org/`

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D., 2001b. Principles Behind the Agile Manifesto.
URL `http://www.agilemanifesto.org/principles.html`

Boehm, B., 1986. A spiral model of software development and enhancement. SIGSOFT Software Engineering Notes 11 (4), 14–24.

Boehm, B. W., 1981. Software Engineering Economics. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., Macleod, G. J., Merrit, M. J., 1978. Characteristics of Software Quality. TRW Series of Software Technology. North-Holland Publishing Company.

Booch, G., October 1995. Object Solutions: Managing the Object-Oriented Project. Addison Wesley.

Cantone, G., Donzelli, P., 1998. Production and maintenance of software measurement models. Journal of Software Engineering and Knowledge Engineering 5, 51–58.

Chrissis, M. B., Konrad, M., Shrum, S., 2003. CMMI: Guidelines for Process Integration and Product Improvement. Addison Wesley Professional.

Chung, L., do Prado Leite, J., 2009. On non-functional requirements in software engineering. In: Borgida, A., Chaudhri, V., Giorgini, P., Yu, E. (Eds.), Conceptual Modeling: Foundations and Applications. Vol. 5600 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 363–379.

Churcher, N., Irwin, W., 2005. Informing the design of pipeline-based software visualisations. In: APVIS2005: Asia-Pacific Symposium on Information Visualisation, volume 45 of Conferences in Research and Practice in Information Technology. ACS, pp. 59–68.

Cohn, M., Ford, D., 2003. Introducing an agile process to an organization [software development]. Computer 36 (6), 74–78.

Collins-Sussman, B., Fitzpatrick, B. W., Pilato, C. M., June 2004. Version Control with Subversion — Next Generation Open Source Version Control. O'Reilly Media.

Currit, P. A., Dyer, M. G., Mills, H. D., 1986. Certifying the Reliability of Software. IEEE Transactions on Software Engineering 12 (1), 3–11.

Cysneiros, L. M., do Prado Leite, J. C. S., 2004. Nonfunctional requirements: From elicitation to conceptual models. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 30 (5), 328–350.

Deissenboeck, F., Juergens, E., Lochmann, K., Wagner, S., 2009. Software quality models: Purposes, usage scenarios and requirements. In: ICSE Workshop on Software Quality.

DuBois, P., Hinz, S., Stephens, J., Brown, M., Bedford, T., 2008. MySQL 5.1 Reference Manual. Oracle Corporation.

El-Wakil, M., 2004. Software metrics - a taxonomy. In: Proceedings of the 3rd International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA04).

Fenton, N. E., Pfleeger, S. L., 1998. Software Metrics - A Rigorous and Practical Approach (2nd edition). PWS.

Fowler, M., 2005. The New Methodology.
URL `http://www.martinfowler.com/articles/newMethodology.html`

García, F., Bertoa, M. F., Calero, C., Vallecillo, A., Ruíz, F., Piattini, M., Genero, M., 2006. Towards a consistent terminology for software measurement. Information and Software Technology 48 (8), 631–644.

Garvin, D. A., 1984. What does "product quality" really mean? Sloan Management Review 26, 25–45.

Gibson, R., 1998. Information systems innovation and diffusion. IGI Publishing, Hershey, PA, USA, Ch. Software process improvement: innovation and diffusion, pp. 71–87.

Gousios, G., Spinellis, D., 2009. Alitheia core: An extensible software quality monitoring platform. In: Proceedings of the 31st International Conference on Software Engineering. ICSE '09. IEEE Computer Society, Washington, DC, USA, pp. 579–582.

Gross, D., Yu, E., 2000. From non-functional requirements to design through patterns. Requirements Engineering 6, 18–36.

Hyatt, L. E., Rosenburg, L. H., 1996. A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality. In: European Space Agency Software Assurance Symposium and the 8th Annual Software Technology Conference, Product Assurance Symposium and Software Product Assurance Workshop, Proceedings of the meetings held 19-21 March, 1996 at ESTEC, Noordwijk, the Netherlands. p. 209.

IEEE Computer Society, 1990. IEEE Standard Glossary of Software Engineering Terminology.

IEEE Computer Society, 1998a. IEEE Guide for Developing System Requirements Specifications.

IEEE Computer Society, 1998b. IEEE Recommended Practice for Software Requirements Specification.

IEEE Computer Society, 2004. IEEE Standard for a Software Quality Metrics Methodology.

IEEE Computer Society, 2005. IEEE Standard for Software Verification and Validation.

International Organization for Standardization, 1999. SANS 14598-1:1999 — Information Technology — Software product evaluation — General overview.

International Organization for Standardization, 2000a. SANS 9001:2000 — Quality management systems — Requirements.

International Organization for Standardization, 2000b. SANS 9004:2000 — Quality management systems — Guidelines for performance improvements.

International Organization for Standardization, 2003. Software engineering - Product quality Part 1: Quality model. Standards South Africa.

International Organization for Standardization, 2005. SANS 9000:2005 — Quality management systems — Fundamentals and vocabulary. Standards South Africa.

International Organization for Standardization, 2007a. Software Engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE. Standards South Africa.

International Organization for Standardization, 2007b. Software engineering — Software product quality requirements and evaluation (SQuaRE) — Requirements for quality of Commercial Off-The-Shelf (COTS) software product and instructions for testing. Standards South Africa.

International Organization for Standardization, 2008a. SANS 12207:2008 — Systems and software engineering — Software life cycle processes.

International Organization for Standardization, 2008b. Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Measurement reference model and guide. Standards South Africa.

International Organization for Standardization, 2008c. Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Quality measure elements. Standards South Africa.

International Organization for Standardization, 2008d. Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Quality requirements. Standards South Africa.

International Organization for Standardization, 2009. Software engineering — Software product quality requirements and evaluation (SQuaRE) — Data quality model. Standards South Africa.

International Organization for Standardization, 2011a. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Evaluation process. International Organization for Standardization.

International Organization for Standardization, 2011b. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. International Organization for Standardization.

International Organization for Standardization, 2013. How does iso develop standards?
URL `http://www.iso.org/iso/home/standards_development.htm`

Jacobson, I., Booch, G., Rumbaugh, J., 1998. The Unified Software Development Process. Addison Wesley.

Khurshid, N., Bannerman, P. L., Staples, M., 2009. Overcoming the first hurdle: Why organizations do not adopt cmmi. In: Proceedings of the International Conference on Software Process: Trustworthy Software Development Processes. ICSP '09. Springer-Verlag, Berlin, Heidelberg, pp. 38–49.

Kitchenham, B., Pfleeger, S., 1996. Software quality: The elusive target. IEEE Software 13 (1), 12–21.

Kruchten, P., December 2003. The Rational Unified Process: An Introduction, 3rd Edition. Addison Wesley Professional.

Leffingwell, D., 2007. Scaling Software Agility: Best Practices for Large Enterprises. Addison Wesley Professional.

Makris, K., 2004. Scmbug manual.
URL `http://files.mkgnu.net/files/scmbug/SCMBUG_RELEASE_0-26-22/manual/html-mu`

Martin, J., 1991. Rapid application development. The James Martin productivity series. Macmillan Pub. Co.
URL `http://books.google.co.uk/books?id=o6FQAAAAMAAJ`

McCabe, T., 1976. A software complexity measure. IEEE Transactions on Software Engineering 2 (4), 308–320.

McCall, J. A., Richards, P. F., Walters, G. F., 1977. Factors in software quality. Tech. rep., Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York.

Michael, C., McGraw, G., Schatz, M., 2001. Generating software test data by evolution. Software Engineering, IEEE Transactions on 27 (12), 1085–1110.

Mills, H. D., Dyer, M. G., Linger, R. C., 1987. Cleanroom software engineering. IEEE Software 9, 19–25.

Mordal-Manet, K., Laval, J., Ducasse, S., Anquetil, N., Balmas, F., Bellingard, F., Bouhier, L., Vaillergues, P., McCabe, T. J., 2011. An empirical model for continuous and weighted metric aggregation. In: Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering. CSMR '11. IEEE Computer Society, Washington, DC, USA, pp. 141–150.
URL `http://dx.doi.org/10.1109/CSMR.2011.20`

Naur, P., B. Randell, E. (Eds.), 1961. Software Engineering: Report of a conference sponsored by the NATO Science Committee.

Østerlie, T., Wang, A. I., 2006. Establishing maintainability in systems integration: Ambiguity, negotiation, and infrastructure. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance, ICSM'06. pp. 186–196.

Paech, B., Kerkow, D., 2004. Non-functional requirements engineering — quality is essential. In: 10th Anniversary International Workshop on Requirements Engineering: Foundation for Software Quality, REFSQ 2004.

Pérez, J., Deshayes, R., Geominne, M., Mens, T., 2012. Seconda: Software ecosystem analysis dashboard. In: 16th European Conference on Software Maintenance and Reeingineering. pp. 527–530.

Phillips, M. (Ed.), August 2006. CMMI for Development, Version 1.2. Carnegie Mellon Software Engineering Institute.

Pierce, B., Jul. 2000. Is cmmi ready for prime time? Crosstalk. The Journal of Defense Software Engineering 13 (7), 21–24.

Poncin, W., Serebrenik, A., Van den Brand, M., 2011. Process mining software repositories. In: 15th European Conference on Software Maintenance and Reeingineering. pp. 5–13.

Pressman, R. S., 2005. Software Engineering: a practitioner's approach, 6th Edition. McGraw-Hill Education.

Richter, D., 1999. Determination of Concurrent Software Engineering Use in the United States. Universal Publishers.
URL `http://books.google.co.za/books?id=lLXWAaOVFLMC`

Royce, W. W., 1970. Managing the development of large software systems. In: Proceedings of IEEE WESCON. pp. 1–9.

Simon, H., March 1956. Rational choice and the structure of the environment. Psychological Review 63 (2), 129–138.

Soanes, C., Hawker, S., Elliott, J. (Eds.), September 2006. The Paperback Oxford English Dictionary, 6th Edition. Oxford University Press.

Staples, M., Niazi, M., Jeffery, R., Abrahams, A., Byatt, P., Murphy, R., Jun. 2007. An exploratory study of why organizations do not adopt cmmi. Journal of Systems Software 80 (6), 883–895.

Stelzer, D., Mellis, W., Herzwurm, G., 1996. Software process improvement via iso 9000? results of two surveys among european software houses. In: Proceedings of the 29th Hawaii International Conference on System Sciences Volume 1: Software Technology and Architecture. HICSS '96. IEEE Computer Society, Washington, DC, USA, pp. 703–.

Stelzer, D., Mellis, W., Herzwurm, G., Oct. 1997. A critical look at iso 9000 for software quality management. Software Quality Control 6 (2), 65–79.

Stevens, W., Myers, G., Constantine, L., 1974. Structured design. IBM Systems Journal 13 (2), 115–139.

Swicegood, T., 2008. Pragmatic Version Control using Git. The Pragmatic Bookshelf.

Taube-Schock, C., Walker, R., Witten, I., 2011. Can we avoid high coupling? In: 25th European Conference on Object-Oriented Programming.

Vetro, A., Morisio, M., Torchiano, M., 2011. An empirical validation of find bugs issues related to defects. In: 15th Annual Conference on Evaluation & Assessment in Software Engineering.

Vlas, R., 2011. A dichotomous stakeholder-centric software evaluation framework. In: Proceedings of Southern Association for Information Systems. SAIS.

Wagner, S., 2007. Using economics as basis for modelling and evaluating software quality. In: Proceedings of The First International Workshop on the Economics of Software and Computation. p. 2.

Wahli, U., Brown, J., Teinonen, M., Trulsson, L., 2004. Software Configuration Managemenat, A Clear Case for IBM Rational ClearCase and ClearQuest UCM. IBM Rational Software.

Whitney, L., 2011. Report: Apple remains king of app-store market | apple - cnet news.
URL `http://news.cnet.com/8301-13579_3-20032012-37.html`

Wyatt, V., Distefano, J., Chapman, M., Aycoth, E., 2003. A metrics based approach for identifying requirements risks. In: 28th Annual NASA Goddard Software Engineering Workshop.