UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

An interpretive case study into the application of software engineering theory.

By Maria Elizabeth Odendaal

Submitted in partial fulfillment of the requirements for the degree
Magister Scientia (Computer Science)
in the Faculty of Engineering, Build Environment and Information Technology
University of Pretoria
October 2010

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

An interpretive case study into the application of software engineering theory.

By Maria Elizabeth Odendaal

Abstract

Even before software engineering was formally defined as a discipline, software projects were notorious for being behind schedule and over budget. The resulting software systems were also often described as unreliable. Researchers in the field have, over the years, theorised and proposed many standards, methods, processes and techniques to improve software project outcomes. Based on allegorical evidence, however, it would seem that these proposals are often not applied in practice. This study was inspired by a desire to probe this general theme, namely of the extent to which (if at all) software engineering theory is adopted in practice.

The core of this research is an interpretive case study of a software project in the financial services industry that ran from end 2006 to mid 2008. I was one of a team of approximately 20 developers, analysts and development managers working on the project, until I left the company in 2009. Results are reported in a two-phase fashion over several themes. Firstly, the literature of recommended software engineering practices relating to a particular theme is reviewed. This is regarded as the "theory". Thereafter, the observations and evidence collected from the interpretive study in regard to the relevant theme is presented and discussed.

The first theme investigated is the notion of "project outcome". Definitions of successful and failed software projects are considered from the perspective of the various stakeholders. Also considered are factors that contribute to project success or failure. After examining how case study participants viewed the project's outcome, it is argued that the project could neither be labelled as a complete success nor as a complete failure. Two areas were identified as problematic: the requirements gathering process; and the system architecture that had been chosen. Improvements in these areas would arguably have most benefitted the project's outcome. For this reason, recommended practices were probed in the literature relating both to requirements engineering and also to software architecture design. The case study project was then evaluated against these recommended practices to determine the degree to which they were implemented. In cases where the recommended practices were not implemented or only partially implemented, a number of reasons for the lack of adoption are considered.

Of course, the conclusions made in this study as to why the recommended practices were not implemented cannot be naïvely generalized to the software engineering field as a whole. Instead, in line with the interpretive nature of the study, an attempt was made to gain in depth knowledge of a particular project, to show how that project's individual characteristics influenced the adoption of software engineering theory, and to probe the consequences of such adoption or lack thereof. The study suggested that the complex and individual nature of software projects will have a substantial influence on the extent to which theory is adopted in practice. It also suggested that the impact such adoption will have on a project’s outcome will be critically influenced by the nature of the software project.

Keywords
Software Engineering; Software Crises; Project Outcome; Requirements Engineering; Software Architecture; Interpretive Research.

Supervisors: Prof. DG Kourie & Dr. Andrew Boake
Department of Computer Science
Degree: Magister Scientia

To Janus, my dearest partner in greatness, for being there from the start,
and Heidi, my little angel, for waiting at the finish.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Acknowledgements

I would like to thank all the Project Evolution participants for their time and cooperation. In particular I'd like to thank Thuli Methula and Donovan Sherriffs for their valuable insight they shared with me, which is not only invaluable to this dissertation but also to me as a software professional. I'd also like to thank Prof. Kourie and Dr. Boake for sharing their knowledge and expertise as well as the encouragement and patient support they gave me the past 5 years.

Content

1 Introduction

Researchers in the field of software engineering often feel that the software industry as a whole would be in far better shape if only the software practitioners working in industry would adopt more of the wisdom and practices prescribed in literature. At the same time it is a common feeling amongst practitioners that these practices, while theoretically sound, are impractical to implement and, as a consequence, they rarely implement them fully. Practitioners often feel academics lack practical experience of what it is like working at the coal face; that they lack knowledge of practical day to day difficulties; and hence that academics generally provide them with theoretical solutions that do not deliver.

In this study we will identify a number of practices recommended in literature to improve the outcome of a software project. The selection of these practices will be informed by considering a number of difficulties encountered on an actual software project. We will then investigate the degree to which these practices were implemented on a specific project, and in the case where they were not fully implemented, interrogate the reasons for not doing so.

The following section will elaborate on the particular research questions we will be addressing in this dissertation. These questions will be addressed by combining a literature study and interpretive case study of a software project on which the writer was a participant. Section 1.2 elaborates on the interpretive research method while section 1.3 introduces the reader to the case study project. The final section of this chapter aims to further clarify the research method used, the reasoning behind the choice of topics included in the study as well as the chosen layout of the remaining chapters.

1.1 Research Questions and Objections

It is the author's belief that the academic community has attempted to address most of the difficulties encountered while building software systems today, by developing and documenting various solutions and recommendations. In other words, it would seem that for any given software engineering problem there appears to be a solution published. At the start of the study, it was also the author's perception that in practice, few of these recommendations are fully or even partially implemented.

The primary research question thus enquires into possible reasons behind the perceived lack of adaption of software engineering theory in practice. It is conceded that this question is premised on the assumption that there is, indeed, a lack of adoption of theory in practice. While this assumption may not be universally valid, there was sufficient prima facie evidence that it is indeed true in the context in which this study took place.

In order to clearly identify the possible reasons for the lack of adoption of software engineering theory in practice, it was decided to limit the scope of this study. This was done by identifying two particular software engineering focus areas where problems were experienced on an industry software project. In order to identify the focus areas it was necessary to consider the following questions: Firstly, what criteria should a project meet in order to be classified as a success or failure; and secondly, what factors contributed to the successful or failed outcome of a project? By identifying key factors that contributed to the projects final outcome the key focus areas became evident. These focus areas are identified in chapter 3 as the requirements engineering process and the systems architecture. The focus areas were then investigated in detail from a theoretical and practical perspective.

Once the focus areas were identified it was necessary to ask further questions: Firstly, what are the recommended software engineering practices related to this focus area; secondly, we asked which of these practices were implemented during the project; and thirdly, we enquired to what degree were they implemented. In the case where the recommended practices were not implemented at all, or were only partially implemented, we were then able to explore the primary question – i.e. to seek reasons for the lack of adoption.

This study will not attempt answer any questions related to the fitness of the recommended practices. It will merely attempt to identify and describe a number of practices related to the focus areas as they become evident through a literature study. It will however attempt to identify the most commonly recommended and mature practices. It will not attempt to speculate on the possible outcome of the case study project had the practices been fully implemented.

1.2 Interpretive Research and Case Studies.

One can make use of various research techniques to answer a research question. One research technique that seemed particularly appropriate is a form of qualitative research, namely interpretive research.

While quantitative research is more concerned with the quantifiable properties of phenomena and their relationships, with the objective of developing mathematical models, theories and hypotheses to explain the phenomena, qualitative research aims to gather in-depth knowledge of human behaviour and the reasons that govern human behaviour. It is however acceptable to include quantitative data in a qualitative study.

Advocates of interpretive research argue that the methods of natural science are inadequate to study social reality, because of the fundamental differences that exist between physical and social artifacts. Those differences are demonstrated by the fact that different persons may attach different meanings to the same object or action.

Klein et al. [17] classify research as interpretive if it is assumed that our knowledge of reality is gained only through social constructions such as language, consciousness, shared meanings, documents, tools and other artifacts. They state that interpretive research does not predefine dependent and independent variables, but focuses on the complexity of human sense making as the situation emerges; it attempts to understand the phenomena through the meanings that people assign to them.

Interpretive research is not without criticism. Garrick [19] voices three categories of doubt relating to interpretive research. Firstly he is concerned about the subjective nature of these studies and their inability to make generalizations based on what can be regarded as factual evidence. His second concern is based on critical theory in that interpretive accounts simply do not go far enough to explain the complex influences upon individual experiences. Lastly he questions the meanings and validity of findings derived through interpretive research. Several authors attempt to provide the interpretive researcher with guidelines and principals to overcome these criticisms and produce sound results. More on these can be found in Appendix 1.

In her book, "*Researching your professional practice, doing interpretive research*", Rednor [16] describes two methods for gathering data, namely observation and interviewing participants using open ended questions to guide the conversation towards the topics of interest. Walsham [21] also suggests supplementing one's data with other forms of field data such as press articles, media and other publications, internal documentation, surveys and so forth.

As data is collected it is necessary to analyze and interpret it in order to answer the research question at hand. To answer the research questions stated in this dissertation the six steps set out by Rednor [16] will generally be followed. This entails ordering the information into topics and categories, coding the various transcripts and then analyzing and interpreting the data.

Seeing that the various research questions all stemmed from the authors experiences on a particular project and the behaviours she observed during this project it seemed fitting to study this project in the form of an interpretive case study. Bensabat et al [23] concurs that case studies are especially well suited to information systems research because they allow the researcher to study the phenomena in its natural setting, and then develop theories from what is observed in practice. They also state that case studies allow the researcher to understand the nature and complexity of the processes taking place – this ties in well with interpretive research. Drake et al. [22] described a number of research areas where case study research could be appropriate. This list is summarized in Table 1-1.

- The study of contemporary phenomena in their natural context, with focus on understanding the dynamics present in a single setting.
- Where research and theory are at their early formative stages.
- Where examination and understanding of the context is important.
- In areas where theory and understanding are not well developed, including areas where the phenomena is dynamic and not yet mature or settled.

Table 1-1 Case study appropriate research areas. (Drake et al.)

A formal definition of case study research reads as follows: "A case study examines a phenomenon in its natural setting, employing multiple methods of data collection to gather information from one of multiple entities (people, groups or organizations). The boundaries of the phenomena are not clearly evident at the outset of the research and no experimental control or manipulation is used." (Bensabat et al. [23]). Upon conducting case study research, the researcher may have little knowledge of the variables of interest and how they are to be measured, and the outcomes of a case study rely heavily on the integrative powers of the investigator.

Case studies may be explanatory, descriptive or exploratory in nature. I intend this case study to be of an explanatory nature, implying that it should give an accurate rendition of the facts, give some alternative explanations for these facts (or behaviours) and then provide a conclusion based on a single explanation that seems most congruent with the facts.

Case studies fit well in both the interpretive and positivist traditions [22]. The positivist researcher is said to remain neutral and objective, her presence not impacting the situation being studied. In sharp contrast the possibility of unprejudiced, zero-impact study is rejected by the interpretive school of thought. Positivist case studies are concerned with the empirical testability of theories in order to discover general principals or laws. They are designed and evaluated in accordance with the natural science model of research: controlled observations, controlled deductions, repeatability and generalizability. Interpretive studies on the other hand are not concerned with the repeatability or generalization of their findings. The value of their conclusion is judged in terms of the extent to which it allows others to understand the phenomena being studied. "Interpretive researchers present their interpretation of other people's interpretations" (Drake et al. [22]).

When designing a case study it is necessary to choose between a single-case, and multiple-case case study. Single-case case studies are said to be most useful at the outset of theory generation and in late theory testing, whereas multiple-case case studies are preferable if one wishes to obtain more general research results and the intent of the research is description, theory building or theory testing. (Bensabat et al. [23]) As my intention is to discover possible explanations for behaviours exhibited in a specific situation, a single-case case study should suffice for this study.

Case study research, as interpretive research, is not without shortcomings, and much has been written to this effect. (Drake et al. [22] Bensabat et al. [23] Yin. [24] Lee. [26]). Fortunately, literature also exists suggesting ways to remedy these shortcomings. More on this subject can be found in Appendix 1.

1.3 Project Evolution

This section aims to provide background knowledge and context around the case study examined in this dissertation as well as the role the author played on this project. Project Evolution is a software project undertaken by the Group Risk division of Discovery Life, a South African life insurance company. The primary objective of Project Evolution was to build a comprehensive, flexible and integrated quoting and policy administration system capable of supporting business objectives.

I joined Discovery Life, in September 2006 as a senior developer in the Group Risk Systems department. At that stage we were supporting two major applications: the quoting system known as Globe used by the New Business department users and the policy administration system used by the Underwriting and Servicing departments' users, known as Globe Admin.

Towards the end of 2006 the status of the business was as follows; they had an Annual Premium Income (API) of R275M, and a 5% market share. They had roughly 1000 schemes with 88 000 members on the books. Their operating costs were roughly double that of their competitors at 18% of API. They lost 25% of new business at quote stage due to the absence of investment products like pension and provident funds. Customer satisfaction was sitting at 72%.

At that stage, the business objective was to increase API to R550M within 3 years, and R1B within 5 years while decreasing the operating cost to 9% of API. They also wanted to increase customer satisfaction to 90%. This was to be done with an innovative new pricing strategy and integration with other Discovery products. They also needed to shorten the time to market for new products to better take advantage of market opportunities.

An extensive audit was performed on the existing systems to determine to what extent they were able to support the business objectives. It was found that the system only provided 54% of the needed functionality at that stage and that it would take at least 10 projects spanning over 30 to 36 months to close the gap. The system also had limited capacity on new products.

Coinciding with these developments, Discovery was planning to launch its new subsidiary, Discovery Invest, an investment product company. The management of Discovery Invest made the decision to acquire a package widely used in industry to administer their investment policies. This package, Compass, was developed by SunGard based in Boston, USA, and used by industry leaders and competitors including Liberty Life.

Because the current systems were not able to support business effectively an investigation was launched to determine the best way to remedy the situation. The options were to fix up the existing systems or to invest in a vendor-supplied solution. Since the investment in Compass was already made by Discovery Invest, the possibility of Group Risk leveraging off this investment and replacing its current administration system with an instance of the Compass package was also investigated.

After some investigation it was decided that Compass, being configured correctly would be a 90% fit to the business requirements, and the decision was made replace the existing policy administration system, Globe Admin, with an instance of Compass. When it came to the quotation system available from SunGard, it was found inadequate and management found in favor of developing an in-house quotation or pricing system for the users in the new business department. Compass also did not provide any reporting functionality, and a considerable amount of development would be necessary to integrate with other systems in Discovery that are necessary for functionality such as payments, and other integrated products like Vitality, Discovery's rewards program.

The configuration of Compass, building of the quote system, development of reports and integrating into corporate systems were classified as different work streams under project Evolution. The systems area was divided into multiple development teams; the Compass team responsible for configuring the Compass system; the Java team responsible for developing the Quotation (also known as the Pricing system) system and integration to other Discovery systems; and the reporting team responsible for the reporting platform in Oracle. A commitment was made to the sponsors that the new system would be delivered end November 2007, giving roughly 12 months to gather business requirements, design, implement and test the system.

As the systems team at that stage lost the majority of developers by the end of 2006 and the remaining developers had no Compass experience, finding the appropriate resources to configure the Compass package proved to be a problem. Initially the decision was made to provide SunGard, the vendor, with the configuration requirements and have them deliver the configured version of the package. This decision was later changed for a number of reasons and Compass developers with experience working for a competing company were hired to do the configuration instead.

The reasons for configuring the Compass package internally included: (1) SunGard was unable to deliver the configured package within the given time constraints. (2) The business analysts had no previous Compass experience which slowed down the requirements gathering process as it was necessary to have very detailed business rules documented to configure Compass properly. Most of the initial versions of the requirements given were from an existing systems perspective which was found not to be implementable in Compass. (3) It was very expensive to outsource the configuration to SunGard, because it was based in the US and payment therefore incurred the penalty of an unfavourable currency exchange rate.

It was only at the beginning of October 2007 that the development team was in place and the requirements agreed upon, leaving a mere 2 months before the initial delivery date. Clearly this date had to be renegotiated, but the sponsors were getting restive and were only willing to push the delivery date out to the end of May 2008. This was admittedly unrealistic but it was made clear that no further allowances would be given.

The delivery date was indeed met with a number of requirements being deferred to the next phase, testing being cut drastically and vast amounts of overtime.

1.3.1 Noteworthy Behaviours and Phenomena Observed

During the implementation and in the months following the initial release of the system there were a number of noteworthy behaviours and phenomena taking place. Since we are trying to identify the areas where the most problems were experienced this section will mostly focus on the negative observations made. It is however important to note that a number of positive observations were also made during the running of the project and during interviews with the participants.

Two noteworthy positive observations include the excitement and joy expressed by the development team as well as the business team over replacing the ailing Globe systems. The development teams were also very happy to learn and work with new technologies. It is also worth noting that within the different development teams, good teamwork and a sense of camaraderie could be seen and many friendships developed.

Most of the negative observations made can be divided into 3 broad categories. Firstly there were a large number of problems experienced during the project related to requirements and project management. The second category includes observations related to the chosen architecture and implementation of the system. The third category includes all those observations that can be described as managerial, human resources or of a sociological nature. It was also observed that testing became a point of great contention during the final stages of the project.

The specific observations made are summarized by category below and then described in more detail in the paragraphs that follow.

Requirements & project management related observations

- Poor quality requirements.
- Constant reworking of functionality.
- Miscommunication and implementation incompatibilities between development teams.
- Inconsistencies between requirements documentation and implementation.
- Erratic release cycles.
- Unrealistic and missed deadlines.
- High number of production defects.

Architectural and implementation related observations

- Architectural degradation.
- Poor system performance.
- Delays and frustration caused by new technologies.
- Inconsistent coding practices.
- Lack of technical documentation.
- Inconsistent user interfaces.

Management & human resources related observations

- Departmental restructuring.
- High staff turnover.
- Large amounts of overtime worked.
- Blurred roles & responsibilities.

- Times of low moral.
- Politics & strained relationships.
- Reversion to old systems.

Poor quality requirements.

Many of the developers commented negatively on the general quality of the user requirements received from the business analysts. The developers working on the quoting system were frustrated mostly by the fact that they would receive requirements to implement that proved to be incompatible with the Compass administration system. Another common complaint was that requirements were generally incomplete.

Constant reworking of functionality.

Adding, removing, changing and adding back of functionality to the system became a common occurrence. This proved to be a source of great frustration to the developers.

Miscommunication and incompatibilities between development teams.

The development teams were always required to work on features simultaneously in order to meet deadlines. It often happened that at the point where the teams were required to integrate their work that incompatibilities were discovered. This usually required a degree of rework by one or more teams and in the extreme cases the business requirements had to be revisited because the requirements given to the quoting team were not compatible with what was possible in the Compass administration system.

Inconsistencies between requirement documents and implemented functionality.

The quoting system delivered was very different from the requirement documents. This was due to several reasons. Firstly certain changes were requested, but never documented. A great deal of necessary functionality was also only uncovered during development as gaps in the requirements became apparent. A number of requirements were omitted as it was not feasible to implement them before the May 2008 deadline or the developers were unable to implement them using the chosen frameworks.

Erratic release cycles.

Due to the fact that many of the functional requirements were deferred to later releases in order to meet the May 2008 deadline, it was necessary to frequently release functionality after the go live date as it became available. The priority of the remaining requirements were frequently changed and often it was necessary to have emergency releases to make functionality available sooner then initially planned. This often required developers to remove features already implemented but not tested from builds in order to fast track the prioritized features.

Unrealistic deadlines & missed deadlines.

Release dates and deadlines were often decided upon by the management team, without input from the developers. This became a popular topic for discussion in the development teams as the feeling was that the deadlines were generally unrealistic. Although the overall May 2008 deadline was met, many of the milestone deadlines were missed leading up to the final release as well as subsequent releases thereafter.

High number of production defects.
A large number of production defects were logged after the system went live. This necessitated not only fixing the defects but also a number of data fixes.

Architectural degradation.
After the May 2008 release, it was found that a number of the architectural constraints had been violated, especially by junior developers.

Poor system performance.
In both the quoting and administration systems, performance was a constant struggle. Loading, processing and persisting of data took longer then acceptable and lots of optimization was required. In extreme cases the optimizations required changes and compromises to the underlying architecture.

Delays and frustration caused by new technologies.
Most of the technology chosen to implement the system with was new and unknown to the developers. This presented the developers with a steep learning curve and they often ran into technical problems that caused delays.

Inconsistent coding practices.
When inspecting the code clear differences in style and quality could be seen amongst the different developers.

Lack of technical documentation.
Very little technical documentation was generated and the documents that did exist were often outdated.

Inconsistent user interfaces.
When navigating through the different parts of the quoting system the screen layouts were noticeably inconsistent.

Departmental restructuring.
People and functions were often moved around. The business analysis team, for example, reported to the business area at the onset of the project, they were incorporated into the systems area towards the middle of the project. Throughout the project and after delivery several resources were moved between the business and systems and teams, and teams were continuously restructured. At one stage the head of systems assumed responsibility for both the systems and business areas.

High staff turnover.

Both the systems and business areas experienced relatively high staff turnover rates. In the business area the high turnover started shortly after project delivery. In the systems area staff turnover was a constant battle from the onset of the project. The average size of the systems team including developers, analysts and managers as well as contracting staff was around 30 people. In the 30 month period between September 2006 and March 2009, 22 systems employees resigned or were asked to resign.

Large amounts of overtime worked.
Working late nights and on weekends became a common phenomena especially in the last months before final delivery with many of the developers working 6 days a week on average.

Blurred roles & responsibilities.
It was often unclear where certain roles and responsibilities lay. Developers and system analysts were required to gather and clarify requirements. Development managers and architects implemented functionality. Business analysts were responsible for testing and training. At one stage the entire systems team were given access to the production system to help clear the backlog of existing business that needed to be loaded onto the new platform, a task usually performed by the business users.

Times of low moral.
At times it was clear that team moral was very low. These times were marked by emotions running high, with project members loosing their temper, shouting at each other and some bursting into tears and threatening to resign.

“Politics” and strained relationships.
Towards the middle of the project, clear divisions between “systems” and “business” became apparent. At times interactions between the two groups were markedly hostile. Examples include systems refusing to undertake work or make changes to requirements without formal signoff. It was also the feeling amongst the systems team that business users deliberately delayed testing and reporting of bugs until shortly before the release date and then refused to provide signoff, causing missed deadlines. Over time a culture of blame also developed with each team using every opportunity that arose to negatively impact the other. The strained relationships also had a negative impact on the running of the business as it started affecting service levels and customer satisfaction negatively. Many of the systems team members were of the belief that business users allowed service levels to slip intentionally to place the new system in a bad light.

Reverting back to old system.
A number of weeks after the Pricing system was released to the users in production they reverted back to using the old Globe system and an Excel Pricing Model to generate quotations. Amongst the reasons given for doing so was that calculating the premiums and costs in the Pricing system took too long. They also cited missing functionality and lack of confidence in the results produced by calculations as further reasons for doing so.

Limited testing.

Large parts of the system were placed in production without undergoing any formal testing. A testing team was in place, but at a late stage in the project it was discovered that no testing plans were available, subsequently the manager in charge of testing was dismissed. The time allocated for testing was also severely cut due to the commitment that development should be completed on time. The testing team members were also very inexperienced and lacked knowledge of what the system should do if functioning correctly.

The case study will be further elaborated on at the end of each relevant chapter, and additional information including organograms showing the various teams, roles and responsibilities can be found in the appendices.

1.4 Research Method & Presentation

The information presented in this study was gathered by means of a literature study and an interpretive case study. At the onset of Project Evolution it was identified as an ideal opportunity to follow and observe the events and behaviours exhibited during an actual industry software project, and then relate that to what was found in literature. The data presented in the case study were collected through observations made by the author being part of the project team as well as through formal and informal interviews held with various project members. Formal interviews were held with the project manager, architect, development manager and systems analyst. Informal interviews were also held with various developers. Additional data was collected by means of questionnaires completed by project participants as well as studying various project artifacts, including requirement document, design documents, project plans and code.

Chapter 2 will discuss a number of key concepts in software engineering including; The Software Crises, The Chaos Report and Fred Brooks' "*No Silver Bullet*" paper. These concepts are often quoted in software engineering literature and serve to illustrate an important point. They illustrate that many of the problems still facing the software engineering field today were hot topics of discussion, even at the birth of the discipline. This makes one wonder: Why after so many years of work and effort are we still experiencing problems developing software? Is it possible for a crisis to last more then 40 years? The *"No Silver Bullet"* paper describes the complex nature of software projects and Brooks' belief that no single solution will provide significant improvements. The validity and accuracy of the Chaos Report is much debated, but nevertheless it does seem to form the foundation and grounds for a lot of research in the field of software engineering. For this reason it is important to be aware of its content.

Chapter 3, 4 and 5 will each in turn, firstly present the appropriate literature study findings followed by the case study findings and conclusions.

Chapter 3 will determine what criteria a project needs to meet to be classified as a success or failure. It will also look at factors that are known to contribute to the successful or failed outcome of the project. It will finally consider Project Evolution and the participants view on the project outcome as well as the contributing factors. These factors will then form the basis of the remainder of the study.

Various problem areas could be identified in Project Evolution, but the choice was made to focus on the requirements process and system architecture in particular. It is a well known fact that the earlier on in the systems development lifecycle mistakes are made, the costlier they are to rectify at later stages. Requirements and architecture both come into play early on in the projects lifecycle and it is mainly for this reason that they were chosen as focus areas for further investigation. The field of requirements engineering and software architecture are also considered to be fairly mature and well documented in literature.

Requirements engineering will be investigated in chapter 4 while software architecture is discussed in chapter 5. In both chapters we will attempt to determine to what degree the recommended practices described in literature were implemented in Project Evolution. In the cases where we found only a limited adoption of recommended practices we will attempt to identify the reasons for not doing so.

Other areas that could have been considered include project management, the human or sociological aspects involved in systems development or systems testing.

2 Crises, Chaos and Werewolves.

Even before the term “software engineering” was coined, the profession faced many challenges and it seems delivering adequate systems within certain cost and time constraints proved difficult from the onset till today. This is evident by examining various contributions over the past four decades.

Chapter outline:

This chapter discusses a number of concepts that greatly influenced the field of software engineering and are referred to throughout this dissertation. The software crises, the findings of the Chaos Report and Brooks’ paper “No Silver Bullet” underline the challenges faced by industry. These are the challenges that researchers are investigating and academia is trying to address.

2.1 The Software Crises

1968 Software Engineering Conference

The terms “software engineering” and “software crisis” were formally described for the first time in 1968 at the Working Conference on Software Engineering held in Garmisch, Germany.

In 1967 the NATO Science Committee established a study group on computer science assigned with the task to assess the entire field of Computer Science. Late 1967 this study group recommended the holding of a working conference on Software Engineering. They chose the phrase “software engineering” deliberately as being provocative, implying that the manufacturing of software should be as disciplined as other established branches of engineering.

The IEEE Computer Society defines software engineering as [39]:

“The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software, and the study of these approaches.”

At the conference, several members were concerned about a rather large gap between what was hoped for and what was actually achieved by complex software systems. They chose to call this the “software crisis” or the “software gap”. Because of the prevalence of this particular concern, this topic was discussed both at a special session and during the plenary session of the conference. [36]

The most common definition of software crises is [35]:

“Software is always over budget, behind schedule and unreliable.”

The conference members had varying views on the seriousness and extent of the problem. Not everyone supported the notion of a software crises, a number of delegates at the conference voiced their concern by pointing out the fact that there already was a number of sophisticated systems running allowing people to perform their work faster and cheaper then ever before. Others indicated that not all areas involved in building software systems were in crises. However many delegates emphasized the fact that even though the software engineering field was making good progress, the demands placed on it by industry simply surpassed what they were able to deliver with the theories and methods of design and production available.

Some of the underlying causes identified by the conference members included the lack of testing, combining research and development into a single production project, lack of management talent, employment of unqualified programmers, incompetent design, inability to accurately estimate time and cost as well as pressure to deliver.

Upon discussing possible solutions, it became clear that it would prove rather difficult to find one simple overall solution to the problems encountered when producing large systems, and comments were mostly made on possible partial solutions such as not committing to deadlines, or using the research content or previously un-encountered problems to gauge the likelihood of disappointment. Emphasis was also placed on the research and study of the software development field to develop best practices and techniques for developing software with a greater chance of success if future.

Dijkstra

Dijkstra later wrote about the software crises in 1972 in his article “The humble programmer”[40]. He recalled that in the early days of programming the hardest part was getting around the severe limitations of the available hardware. He described the available hardware as a pinching shoe, and the programmer as a clever, puzzle-minded individual capable of using clever tricks to get around the hardware limitations. It was believed that as the capabilities of hardware would improve, programming should become increasingly easier to do. Of course the opposite proved true. The capabilities of hardware grew by as Dijkstra puts it a factor of more then a thousand and society’s ambition to apply its machines grew proportionately. This increase in computing power made solutions feasible that the programmers of the day dared not dream about. The fact that the limited capability of hardware was solved but not the problem of scale and complexity resulted in the software crises.

Dijkstra stated that those who want really reliable software must find the means to avoid the majority of bugs to start off with. He was convinced that three major conditions must be fulfilled for this to happen. Firstly the world at large must recognize the need for change; secondly the economic need must be sufficiently strong; and thirdly, the change must be technically feasible.

In conclusion Dijkstra summarized the lesson that he wanted to convey as: "We shall do a much better programming job, provided that we approach the task with full appreciation of its tremendous difficulty, provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as Very Humble Programmers."

2.2 No Silver Bullet

In is essay, "*No Silver Bullet*", written in 1986, Brooks draws on the similarities between the nature of many software projects and werewolves.

"Of all the monsters who fill the nightmares of our folklore, none terrify more then werewolves, because they transform unexpectedly from the familiar into horrors. For these, we seek bullets of silver that can magically lay them to rest." [41]

He indicates how innocent, straightforward software projects often turn into monsters of missed schedules, blown budgets and flawed products, and that we too seek magical solutions to these problems. Brooks states that he foresees no such a single magical solution or silver bullet on the horizon for at least the next decade, neither of technological or managerial nature.

Firstly it is important to understand the way Brooks classifies or divides the task involved in software construction into essential tasks and accidental tasks. Essential tasks were originally defined as "the fashioning of the complex conceptual structures that compose the abstract software entity", while accidental tasks as "the representation of those abstract entities in programming languages and mapping them onto machine languages within space and speed limitations". This classification of tasks was misunderstood by many, and Brooks clarified the intended meaning in a new chapter in the anniversary addition of The Mythical Man Month published in 1995, entitled No Silver Bullet Refired [42], Brooks elaborates, stating that he did not intend accidental to be interpreted as by chance or misfortunate, but rather as incidental or appurtenant. Put simply, the essential tasks relate to the mental crafting of the conceptual construct, and accidental tasks to the implementation process.

Brooks believed that most of the past gains in productivity came from addressing accidental difficulties related to accidental tasks such as severe hardware constraints and awkward programming languages. Central to his argument that no single breakthrough will yield an order of magnitude improvement in productivity, is the argument that unless the accidental tasks make up 90% of the development effort and it was possible to shrink the effort required to zero time, the essential difficulties will have to be addressed to obtain meaningful gains. It is the nature of these essential tasks, that he states as the reason for software development to always remain difficult, making it impossible to have a single silver bullet.

Brooks substantiates his statement by examining both the nature of the software problem as well as the proposed solutions. Brooks believed that the hardest part of software development had nothing to do with the labour of representing the conceptual construct but rather the specification, design and testing thereof. He describes a number of inherent properties of modern software systems, which makes the software engineering process more laborious, including: complexity, conformity, changeability and invisibility.

According to Brooks many of the classical problems relating to software engineering are due to the essential complexity and the non-linear increases in complexity as the size of the system is increased. Complexity increases the difficulties in communication amongst team members, which in turn could lead to product flaws, schedule overruns etc. It is clear that complexity lends itself not only to technical problems but also to managerial problems. Software systems are not only complex but in most cases they are required to conform to some or other requirement usually imposed by the environment in which they are to operate. These conforming requirements are not always necessary but merely reflect the preferences of different individuals. Because software is "infinitely malleable" as Brooks puts it, it is always subject to change. Of course each time software is subjected to change, the possibility of introducing more complexity and the accompanying difficulties is increased. Lastly, throughout its life cycle software remains mostly invisible. Even though it is possible to depict many aspects of software systems visually, they remain too complex to portray concisely. This not only impedes the process of design within one mind, but severely hinders communication amongst several minds.

Brooks mentions three so called "breakthroughs" that improved productivity, but he indicates that all of them addressed accidental difficulties. These included: high-level languages, time-sharing and unified programming environments.

Looking to the future, Brooks distinguishes between a number of developments upon which hope of a silver bullet was placed. Most of these attacked the accidental problems according to Brooks and he did not see much promise in them. These included:

- Ada and other high-level language advances.
- Object-orientated programming.
- Artificial intelligence.
- Expert systems.
- Automatic programming.
- Graphical programming.
- Program verification.
- Environments and tools.
- Workstations.

Brooks did not have much faith in the above developments, but he did suggest that the following might hold some promise, as he believed that these address the essential tasks of the software effort:

- Off the shelf software.
- Rapid prototyping to establishing software requirements.
- Incremental development.

- Identifying and developing of great designers.

Brooks wrote this essay in 1986. Since, many critics as well as ardent followers have been interpreting it in a very negative or pessimistic manner, translating Brooks' message as that of despair: that indeed there is no hope for substantial improvements in the field of software development. Many have argued against this view, and attempted to put their own inventions forward as the so called silver bullet. Brooks makes it clear in a his 1995 "*No Silver Bullet Refired*" chapter of the "*Mythical Man Month"* that he still believes that no single solution by itself would yield a magical ten fold improvement in productivity, but that he in fact remains optimistic for incremental improvements.

2.3 The Chaos Report

In 1994, many years after the software crisis was first described in 1968 the Standish Group undertook a research project to gauge the state of the software industry focusing on the scope of project failures; the major factors causing software project failures as well as key ingredients that can reduce project failures. [38] The sample size of the study was 365 respondents, representing 8 380 applications across an unknown number of small, medium and large companies across different industries. What they termed key findings were collected during personal interviews and surveys with IT executive managers. In addition they also conducted focus groups to provide qualitative context for the survey results. Subsequent studies were also performed.

In the Chaos Report projects are classified as successful, failed or challenged. Successful projects are those that are completed on-time, on-budget, with all the features and functions originally specified. Challenged projects are completed but without meeting all the time, cost and feature requirements. Failed projects are those that are cancelled sometime during the development cycle. Figure 2-1 summarizes the findings of Standish Group in the year 1994, 2000 and 2003 regarding project outcomes amongst those surveyed.

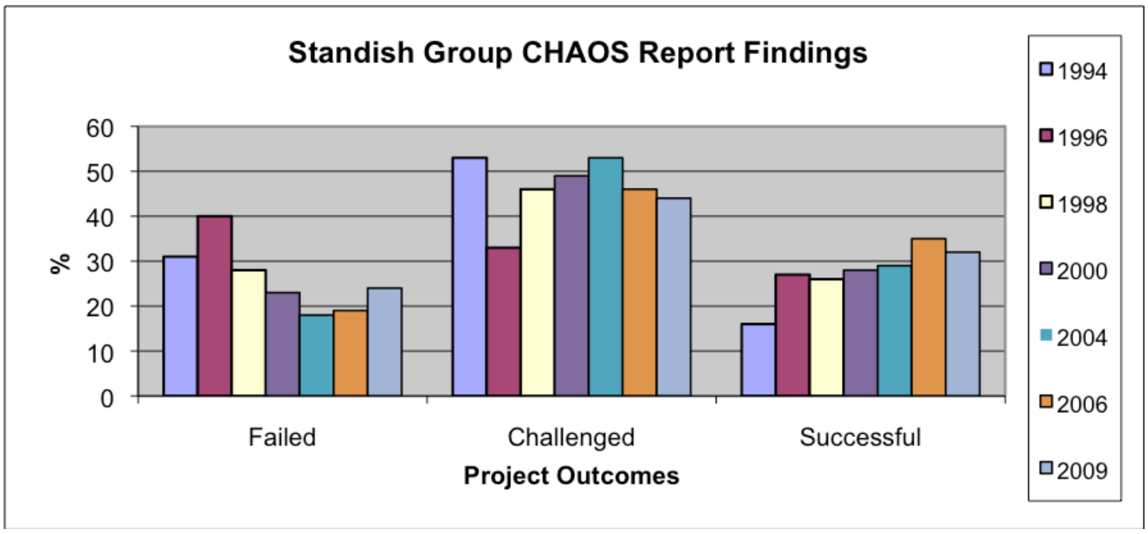


Figure 2-1 CHAOS Report Overall Outcomes

The 1994 report stated that 31.1% of projects will be canceled before they are completed. 52.7% of projects will be challenged and cost 189% of their original estimates. They reported that only 16.2% of projects will be completed on-time and on-budget. However, in larger companies those projects will only contain about 42% of the originally proposed features, while in smaller companies this increases up to 74.2%.

The study also attempted to understand why so many projects fail. The table below summarizes the factors reported to contribute to success or failure respectively along with the frequencies that were reported.

Project Success Factors	**%**
1. User Involvement	15.9
2. Executive Management Support	13.9
3. Clear Statement of Requirements	13
4. Proper Planning	9.6
5. Realistic Expectations	8.2
6. Smaller Project Milestones	7.7
7. Competent Staff	7.2
8. Ownership	5.3
9. Clear Vision and Objectives	2.9
10. Hard Working, Focused Staff	2.4
Other	13.9

Project Challenged Factors	**%**
1. Lack of User Input	12.8
2. Incomplete Requirements & Specifications	12.3
3. Changing Requirements & Specifications	11.8
4. Lack of Executive Support	7.5
5. Technology Incompetence	7
6. Lack of Resources	6.4
7. Unrealistic Expectations	5.9
8. Unclear Objectives	5.3
9. Unrealistic Timeframes	4.3
10. New Technology	3.7
Other	23

Project Failure Factors	**%**
1. Incomplete Requirements	13.1
2. Lack of User Involvement	12.4
3. Lack of Resources	10.6
4. Unrealistic Expectations	9.9
5. Lack of Executive Support	9.3
6. Changing Requirements & Specifications	8.7
7. Lack of Planning	8.1
8. Didn't Need It Any Longer	7.5
9. Lack of IT Management	6.2
10. Technology Illiteracy	4.3
Other	9.9

Table 2-1 1994 CHAOS Report Success/Failure Profiles

Opposing opinions

Robert Glass [35][37] strongly opposes the idea of the software field being in crises and questions the findings of the Chaos Report. He argues that we would not be in the so called "information age" if the majority of software projects were as challenged as indicated by literature. Glass offers several reasons why researchers and industry alike echo the notion of a software crises and the popularity of the damning statistics such as those offered by the Standish Group. These reasons are mainly based on the benefits that can be gained from these dire predictions - such as vendors selling products or services that offer the cure, researchers gaining funding towards finding the cure; or academics gaining stature by publishing papers describing the cure.

In a 2003 paper, Molokken and Jorgensen [43] reviewed a number of surveys done on software cost estimation and drew the following conclusions. Firstly most projects surveyed (60-80%) encounter effort and/or schedule overruns. Secondly even though the Chaos Report described an average cost overrun in the area of 89%, the average overrun found in other surveys were between 30-40%.

Several authors challenge the accuracy and relevance of many studies conducted on the stance of the software industry[43]. Glass [35] demonstrates this by referring to the Government Accounting Study (see section 3.4.1. for more detail). This study indicated a failure rate of 98%. However it focused on projects already in trouble. Placing the study in perspective it is hardly surprising that the majority of projects in trouble fail, although this did not stop authors and speakers repeating these findings without mentioning the context. As for the Chaos Report, The Standish Group does not provide much insight into the means by which the study was conducted and how they deduced the results that they published. Even though the Chaos Report has never been peer reviewed, the typical reaction by many industry practitioners is that of agreement, and this is surely of notable importance.

2.4 Conclusion

Considering these publications we are able to see a number of commonalities even though they were published in the late 60's, 80's and 90's respectively.

Firstly time, cost and quality have always been a concern when developing software systems. Secondly the degree to which we should be concerned has always been debated. Lastly there are a number of common causes or aggravators of project failure given in all these publications. These include complexity, change, pressure to deliver and lack of skills, which include management as well as technical skills.

The question now is: How does the Software Crises, No Silver Bullet and the Chaos Report relate to this study and our research questions? One of our secondary questions is: Is there reason for concern? Looking at these artifacts it is easy to conclude that the software industry, albeit improving, is indeed still facing many challenges, or as some would even put it, in crises, which definitely indicates a cause for concern from a literature study point of view.

But is it possible for a crisis to span over several decades? The very definition of crises refers to an unstable situation of extreme danger or difficulty, or a crucial stage or turning point in the course of something – not to a lengthy period of time. Does the software industry really still face the same difficulties it did back in 1968? Is it possible that we have faced and overcame several crises over the past 40 years, the solution to each preceding crisis clearing a path for the next? Dijkstra deemed this to be a possibility when discussing the inadequacies of available tools to aid software development. Quoting from his 1972 paper, the Humble Programmer [40] “Programming will remain very difficult, because once we have freed ourselves from our circumstantial cumbersomeness, we will find ourselves free to tackle the problems that are now well beyond our programming capacities.” The fact remains: there are numerous challenges facing the software development profession, many projects are affected negatively by these challenges, and there is indeed cause for concern.

These publications also give us insight into the areas of software engineering where improvements could be beneficial. We are able to identify the areas of requirements, design and testing to be of key importance when it comes to developing systems, and investigating ways of improving quality in these areas are believed to improve the overall quality and outcome of the system.

3 Project Outcome

The Chaos report stated that up to 70% of software projects are challenged or failed, with cost overruns of up to 187%, while other studies report far fewer failures and smaller cost overruns in the region of 30%. [38] [43] The very definition of project success of failure is a much debated topic in software engineering and project management, and this could be one explanation for the discrepancies found between the results of different studies aiming to answer essentially the same questions. This chapter aims to highlight the different definitions of project success and investigate some of the common factors that cause projects to fail or succeed.

Chapter outline:

3.5.4 Conclusion

Firstly we will look at some definitions of project success and factors that go into the evaluation of project outcomes in section 3.1. Section 3.2 describes a number of factors as found in literature that are said to contribute to the success or failure of software projects. Lastly we will investigate the influence of industry type on project outcome in section 3.3 and look at some interesting research findings on the topic in section 3.4. The final section will discuss how project success was defined on the case study project, Project Evolution. We will report on whether the participants believed it to be a success or failure as well as what factors they believe contributed to this outcome.

3.1 Defining Success & Failure

“The project is considered an overall success if the project meets the technical performance specification and/or mission to be performed, and if there is a high level of satisfaction concerning the project outcome among key people in the parent organization, key people in the project team and key users or clientele of the project effort.” [30]

In order to judge the outcome of any project we need some or other definition of success or failure. We need criteria against which we can measure the fruits of our labour. The above definition of project success touches on various important aspects of defining project success, including what functionality and performance is required, participant satisfaction and various stakeholders. In the following section we discuss these and other elements that various authors deem important when defining project success or failure.

3.1.1 Functionality, Time, Cost and Quality.

The most popular measure of project performance is undoubtedly whether the project is completed, delivering all the requested functionality within the allocated time, at the target cost and pre-agreed level of quality [30] [31] [32]. The fact that these criteria are very hard, almost impossible, to measure accurately and precisely makes this a very impractical measure of project success. It is also possible to only miss one or few of these targets. For example, would a project that overruns its schedule but comes in under budget be judged as a success or failure? Perhaps it would depend on the importance of delivering on time or exceeding the available budget?

We often refer to the “Iron Triangle” of cost, time and scope. Agarwal et al. [30] suggest that assessments fabricated around the Iron Triangle reflect the success or failure of the implementation process rather then the project as a whole. In their exploratory study to find success indicators [30], they interestingly found scope (which includes functionality and quality) to be rated as the most important success criterion, with functionality being preferred over quality. This is an interesting finding as it is well known that with project schedules and budgets often set in stone, many project teams often have no other choice but to drop or defer functionality to future releases to achieve target dates. Could this strategy perhaps be doing more harm then good to the overall outcome of the project?

Wateridge [32] quoting Kerzner, adds three interesting criteria to the standard time, cost and scope definition of project success, they include that the project must be completed with the minimum or mutually agreed upon scope change; without disturbing the main flow of work for the organization; and without changing the corporate culture.

3.1.2 Customer Satisfaction.

It might be due to the subjective nature of many of the available success measures that several researchers describe project success in terms of customer satisfaction and welfare. Shenhar et. al. [31] quoting Baker et al, (1988) extend the notion of customer satisfaction to include the level of satisfaction of four different groups of shareholders including the customer organization, the developing organization, the project team and the end-user.

3.1.3 Stakeholder Perspectives and Perceptions.

Another reason why the definition of project success is so rarely agreed upon is the fact that different stakeholders of the project have different success criteria against which they evaluate the project [31]. Wateridge [32] deems it crucial to ensure all project members share the same evaluation criteria of project success. If this is not the case one or more project members will inevitably perceive the project as a failure if their expectations are not met.

In Wateridge's study [32] project success criteria was evaluated from different project participants' points of view. The six most important criteria across all respondents and all projects that came out of the study were: (1) Meets user requirements; (2) Achieves purpose; (3) Meets time scale; (4) Meets budget; (5) Happy users; (6) Meets quality. Although this was very much as predicted, the interesting observation was the different emphases placed on the success criteria by project managers and systems analysts and users on the other hand. The results are depicted in Figure 3-1.

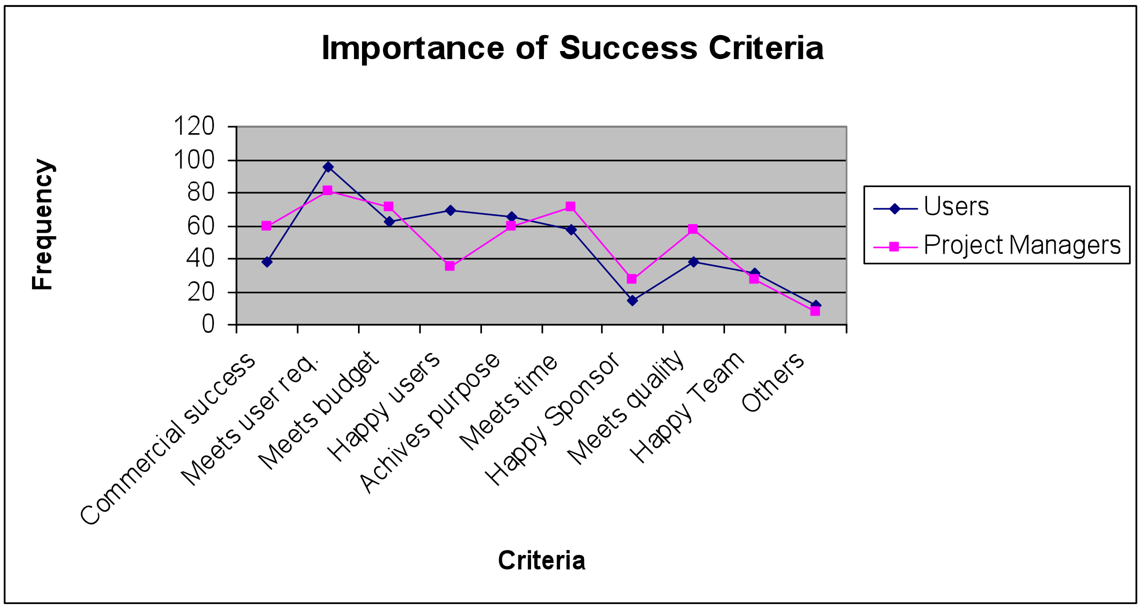


Figure 3-1 Criteria for success - User vs. Project Manager (Adapted from How can IS/IT projects be measured for success. Waterage 1998)

The fact that all parties placed great importance on meeting requirements, but project managers and system analysts in contrast with users did not place much importance on user happiness indicated that they generally implement their interpretation of requirements, which does not necessarily coincide with the user's interpretation. The fact that users require the delivered system to efficiently perform their jobs in the long run, would explain their emphasis on meeting requirements and user happiness or satisfaction.

While users also placed a certain amount of importance on meeting budget and time lines, they would generally not see a project as failed if it costs somewhat more or is delivered later then planned. Project managers on the other hand place considerably more importance on these short term measurements of success, possibly because it is these criteria that are mainly used to evaluate *their* success as project managers.

Wateridge concluded that project managers generally focus on short term criteria such as time and schedule, or the 'process' and users in contrast place greater importance on long term or 'product' success criteria, and that there is a need to address both 'process' and 'product' criteria when evaluating project success. He stresses the importance of having a shared understanding and agreement on what criteria are to be used to measure project success by all at the outset of the project, and the importance of having a tradeoff when criteria conflict.

Procaccino et al. [33] also investigated the definition and criteria of success, but from the developers' point of view. They are of the opinion that due to the fact that developers are at the core of software development, responsible for designing, creating and modifying software systems as well as the fact that they represent the single largest cost of software projects, it is crucial to include their perspective in arriving at an insightful definition of software success. In their study they addressed three areas that influence the outcome of the project: they included Process-related Components, Work-related Components and Project-related Components. The following tables show the importance developers placed on these components when surveyed.

	Ranked Components		
Ranking	**Process-related**	**Work-related**	**System/Project-related**
1	Requirements that are clear and understood by the team	You had sense that you delivered sufficient quality	Requirements of customers/users were met by complete system
2	Team sufficiently skilled	You had sense of achievement	Final system worked as intended
3	Customers/users and team good relationship	You were provided with enough independence/freedom	Project was delivered when needed by customers/users
4	Customers/users provide team with feedback	Your work was technically challenge	Final system consisted of solid, thoroughly tested code
5	Customers/users have realistic expectations	You learned something new	Users found system easy to use
6	Sufficient access to business/technical experts		Project was completed on time
7	Team included in decision making		Project was completed
8	Defined development methodology		Project was completed at cost affordable to the customer/organization
9	Requirements are excepted by team as realistic/achievable		Project was completed within budget
10	Team small and high performing		
11	On-going/realistic evaluation by project manager		
12	Project manager provides team with feedback		
13	Lack of scope creep		
14	Team able to negotiate changes		
15	Team does not feel pressured		

Table 3-1 Process-, Work- and System/Project-related components ranked in descending order Procaccino et al. [33]

Glass [45] writes about Kurt R. Linberg's research where he came across a curious phenomenon in which the developers on a project that overshot its budget by 419% and its schedule by 193% described it as a huge success. When asked why they classified the project as a success they commented that firstly it was working as it was supposed to, secondly developing it was a technical challenge and thirdly their team was small and high performing.

Agarwal and Rathod [30] also mention the developers view on project outcomes. They stated that developers link the success of a completed project to quality and tie uncompleted projects with what was learned that can be applied to future projects. It is their opinion that developers link the success of their efforts with the work satisfaction they get through creativity and learning while working on the project.

3.1.4 The Time Dimension.

It has also been said that there is a time dimension to project success [30]. The outcome of the same project might be judged differently at different times. It might for example be judged as a failure directly after delivery, perhaps because it overshot its budget, but then be seen as a huge success a year later because of substantial benefits that it delivered to business.

Shenhar et al. [31] wrote that the success criteria on which the project outcome will be judged should be agreed upon upfront. They defined 4 dimensions against which project success can be evaluated, namely project efficiency, impact on the customer, business and direct success and finally preparing for the future. The first of their 4 project success evaluation dimensions, p*roject efficiency* is a short term measure that indicates how well the project process was managed. How well the project met its schedule and budget is included in this dimension, but could also include measures such as reliability, safety or number of changes necessary before final release. Their second success dimension, *impact on the customer*, includes the level of customer satisfaction regarding the degree to which performance measures, functional requirements and technical specifications were met. The third dimension*, business and direct success,* addresses the impact that the end product has on the organization. Did the project have the desired impact on sales, income, profitability and market share? Finally the fourth dimension Shenar et al. described, *preparing for the future*, addresses the preparing of organizational technology and infrastructure for the future. All four dimensions can only be evaluated at different times. The first dimension can only be evaluated in the very short term, directly after the project is completed. The second dimension can only be evaluated a short time after the customers had a chance to use the system. The third dimension can only be reliably assessed after enough time has passed to process a significant number of sales, and the fourth dimension only in the long term, probably three to five years.

3.1.5 Project Failure

Defining failed projects seems to be no less challenging then defining successful projects. Yeo [44] describes a number of believes of information system failure by citing a number of authors.

Firstly, Yeo [44] quoting Sauer, proposes that systems should be considered failures only if there is a development or operation termination. As long as the system acts on its environment in such a way to obtain the necessary resources and support to continue its operation, it cannot be considered a failure.

Yeo [44] also puts forward a definition of IS failure described by Flowers, whereby a system is considered a failure if any of the following situations occur: (1) when the system as a whole does not operate as expected and it's overall performance is sub-optimal; (2) if, on implementation, it does not perform as originally intended or if it is so user-hostile that it is rejected by users and under-utilized; (3) if the cost of the development exceeds any benefits the system may bring throughout it's useful life; (4) due to problems with the complexity of the system or the management of the project, the information system development is abandoned before it is completed.

Yeo lastly describes the four major notions or categories of IS failure as originally defined by Lyytin and Hirscheim as follows: (1) Correspondence failure, when the systems design objectives are not met; (2) Process failure, this occurs when a IS cannot be developed within an allocated budget, and/or time schedule; (3) Interaction failure, the level of end-user usage of the system is suggested as a surrogate in IS performance measurement; (4) Expectation failure, the inability of a system to meet it's stakeholders' requirement, expectations, or values.

Boehm [51] interestingly argues that a terminated project does not necessarily equate to a failed project. Boehm states that many projects are properly started, well managed and then properly terminated before completion because their original assumptions had changed. He describes a danger in labeling all terminated projects as failures as project managers might be tempted not to promptly terminate projects that are clearly wasting resources with no hope for return on investment. He attributes project managers' hesitation to terminate projects mainly to fear of the stigma related to managing a "failed" project and the negative impact this might have on the project manager's career. Boehm refers to the top 10 reasons given for project failure in the Chaos Report (see 2.3) and concludes that most of these reasons may not necessarily be attributed to bad project management efforts with the exception of: lack of executive support; lack of planning; lack of IT management; technology illiteracy. In the case of lack of executive support, Boehm believes the project managers failed to verify unrealistic assumptions about executive support in the majority of failed projects. As for lack of planning, these projects are usually terminated when it is discovered that project managers have no idea where in the project they find themselves. Projects that fail due to technology illiteracy are usually projects that should have never been started and projects that failed due to lack of IT management are mismanaged by definition. As stated before, the six remaining reasons for project failure given in the Chaos Report could be present even on well managed projects.

Boehm [51] includes the following guidelines to successfully terminate a project. Firstly he suggests marketing project products. In so doing the project manager is able to create a realistic expectation of deliverables. Marketing practices include activities such as conducting user satisfaction surveys; holding internal technology fairs; ensuring user participation with project reviews and prototyping; clearly stating the impact changes might have on the project including possible project termination. Secondly he recommends using risk management and incremental commitment. By using a risk acceptance approach he states that a certain fraction of project terminations become an acceptable way of coping with an uncertain world. He advocates the risk driven spiral model as a process framework to provide incremental commitment one cycle at a time, allowing for termination when the risk becomes too high. Thirdly he urges for the use of architecture review boards to review the architecture vs. business feasibility in order to determine project continuance, redirection or termination. Boehm's fourth recommendation for determining if a project should be terminated is to frequently monitor business assumptions. If the project was commissioned on certain critical assumptions, such as being first to market, that no longer hold true, the stakeholders need to be notified and the project feasibility reassessed. Lastly Boehm stresses the importance of not equating project termination with project failure because for as long as project termination threatens the career of the project manager, he will be tempted to continue with projects that should die.

3.2 Common Causes of Project Success and Failure

One of the main reasons cited for project failure is complexity[41]. Project teams often underestimate the level of complexity they are dealing with. Xia et al. [46] point out two reasons why IS projects are inherently complex. Firstly they do not only deal with technical issues but also with organizational factors largely beyond the project teams control. Secondly, both the technical and organizational environment is constantly subject to change. These constant changes make it increasingly difficult to estimate and manage requirements and specifications.

Brooks [41] also considers complexity as an essential property of software systems. The reasons he gives for this include: (1) no two parts are alike; (2) programs have a large number of states, making conceiving, describing and testing them hard; (3) when scaling up a software entity the interactions amongst elements increase more then linearly. He also states that complexity creates both technical and management problems.

Various contributing causes and factors related to both project success and failure have been recorded in literature. Table 3-2 and Table 3-3 have been compiled from a number of literature sources. They summarize a number of success and failure factors respectively and classify them into a number of categories named by the column headings. These categories will now be discussed in more detail in the sections to follow.

3.2.1 Requirements Project Vision, Objectives and Scope

"Requirements deficiencies are the prime source of project failures." – Glass' law [52]. It seems that no book, paper, report or study published on the topic of project success or failure ever fails to mention the need for good requirement specifications or the lack thereof as a contributing factor or cause of the project outcome. In the original Chaos Report [38], incomplete requirements were rated as the number one and two reasons for failed and challenged projects respectively. When it came to successful projects, clear requirements were listed as the third most important contributing factor.

Having a clear project vision and objectives as well as a well defined scope is often described as an important contributing factor to project success. On the other hand, projects that commence without well defined objectives and vision are usually seen as doomed, as are projects that experience continuous scope creep during the lifespan of the project.

When we remind ourselves that project outcome is measured differently by different stakeholders (see 3.1.3) it perhaps makes sense why such importance is placed on the vision, objectives, requirements and scope specification. Having all of these specified clearly and upfront could serve as a communication tool to all the different stakeholders and manage expectations amongst them. When each stakeholder understands exactly what is expected of him or her, as well as what he or she can hope to gain from the venture, they are less likely to be disappointed by the final outcome.

3.2.2 Management and Sponsor Support

It is a common belief that the technology and tools exist to solve almost any software problem, yet many software projects are still challenged. Many of the contributing factors to both project success and failure are often attributed to management issues, or to what is popularly referred to as "politics".

It has been shown that projects that enjoy commitment and support from sponsors and management usually have a greater chance at success then do projects that do not. Factors under control of management, notably having unrealistic deadlines, usually have a far greater impact on the outcome of the project than the technical factors under the development team's control. It is thus important that management has a good understanding of industry best practices and encourages their implementation while paying attention to people or soft issues.

3.2.3 Project Management

Project management can be defined as the facilitating, planning, scheduling and controlling of all activities that must be done to achieve project objectives. [53] The project manager clearly plays a very important and central role in software development. It is thus not surprising that next to requirements related factors, the most success or failure factors cited in literature are related to the project manager and/or her duties. These include planning, managing project scope, managing expectations, cost and schedule estimations, risk identification and mitigation, progress tracking and more, as can been seen in the factor summary and classification tables.

3.2.4 Communication and Working Relationships

Software development requires close collaboration between development team members, management and the customers. This is mainly due to the inherently complex nature of software. This need for collaboration between all the different parties involved in the software development effort requires good communication and working relationships to work. It is for this reason that communication or the lack thereof is often cited as contributing factors to project success or failure respectively. With good communication the users are able to convey their needs effectively and realistic expectations can be set amongst all stakeholders. In turn, having a good understanding of what is required also limits the amount of changes required at a later stage when misunderstandings are discovered. Having good working relationships amongst all parties involved encourages risks to be identified and mitigated early, avoiding last minute catastrophes that typically push up costs or delay delivery. In the likely event of a deadline being missed, or budget being overshot, having a good relationship between the development team and the sponsors is especially important in order for the parties to reach a compromise and for the project to ultimately be evaluated as successful.

3.2.5 Knowledge, Skills and Resources

Software systems are essentially a product of intellectual efforts. Having resources with the appropriate knowledge, skills and experience is absolutely essential to the success of the project. It is equally important that the development team has sufficient access to business or domain experts as well as any other resources that they may need to perform their duties. Much has also been written on the topic of retaining skilled resources and the negative effects high turnover has on project outcomes. Whenever a project member is lost not only is valuable knowledge often lost, but additional time is lost and additional costs are incurred to find and train a replacement.

3.2.6 Technology and Tools

Although it can be argued that most of the time the outcome of a project is mostly contributed to process or people related factors, technology could potentially have a big impact. Using a very new technology unfamiliar to the development team is one particularly big risk. It is very likely that the development team will be unable to foresee many of the possible challenges that they might face using a new technology, which in turn could affect the schedule negatively. The inappropriate choice of technology is another factor often mentioned when projects fail as is technology that requires many customizations to be suited to the organization’s needs. Sometimes the choice in technology is adequate but the development team just lacks experience in the use of the particular technology. On the other hand, providing the team with the proper tools has been noted as having very positive effects on productivity.

3.2.7 Risk Management

Most projects face a number of risks. It is vital to analyze and evaluate these risks both before the project is started and as it progresses. This is important for the project sponsors to make informed decisions on the feasibility of the project. Having prior knowledge regarding risks also enables the project manager to react proactively with risk mitigation strategies and allows her to monitor areas of concern more closely.

3.2.8 Estimates and Scheduling

“Most cost estimates tend to be too low” – DeMarco-Glass law [52]. From the literature study of the definition of project success it was clear that most project outcomes are judged by whether they were completed on time and within budget. It thus goes without saying that bad cost estimates or unrealistic schedules will doom any project, and hence it is also one of the most cited factors. Cost and schedule estimates bear a close relationship to project scope and planning. Having bad estimates often necessitate scope or feature cuts which could have a negative effect on the perception users have on the system. Another alternative is to work longer hours or add additional resources to the project, but this could push up costs or jeopardize quality and as Brooks stated, adding manpower to an already late project makes it later [42].

3.2.9 Design

Having a good design is a system’s best defense against change, and since a certain amount of change is inevitable, having a good design that is both stable and flexible is of utmost importance. Evaluating the design against the requirements by making use of architecture evaluation methods is one way to increase the chances of good design. This will be discussed further in following chapters.

3.2.10 Process

Many of the known reasons projects fail can be addressed by various activities in the software development process. For example, following a well defined requirements engineering process during requirements elicitation will greatly improve the quality of the requirements which in turn gets the project started on the right foot. A great number of software development methodologies have emerged over the years, and following one that has been tailored to the needs of the organization has been shown to improve the development effort. Making use of an evolutionary life cycle together with prototyping has particularly been linked to project success. “Mature processes and personal discipline enhance planning, increase productivity, and reduces errors.” – Humphrey’s Law [52].

3.2.11 Change

“A system that is used will be changed” – Lehman’s first law [52]. There are a number of reasons why numerous changes need to be made during software development. A number of changes can be related to errors or omissions made during requirements gathering but even if much care and effort is put into the requirements engineering process the business environment in which the software is to operate is constantly subject to change. This is due to the fact that in today’s business environment organizations constantly need to change and adapt to remain competitive or respond to other external factors such as legislation changes. The very nature of software, being mostly an intangible product further makes it very vulnerable to change as it is perceived to be easily changeable. Changes in the underlying technologies used often mandate changes to the system too, in turn threatening the project schedule and or budget.

3.2.12 Testing

Interestingly when studying the literature for factors contributing to project success or failure one does often come across testing as one such factor. It is usually one of the last steps performed in most software development life cycle methodologies but particular emphasis is placed on testing in agile development methodologies such as XP. In fact, in the XP process, testing is required as one of the first steps in so called test driven development, where a test case needs to be prepared before any implementation is written. In XP all functionality requires automated test cases. The reasoning behind complete and automated test coverage is to allow refactoring without the fear of breaking existing functionality. This allows flexibility while maintaining quality.

The individual success and failure factors included in the above mentioned categories, as found in literature is now listed in Table 3-2 and Table 3-3.

Success Factor	**Requirements, Objectives, Vision & Scope**	**Management & Sponsor Support**	**Project Management**	**Communication & Working Relationships**	**Knowledge, Skills & Resourcing**	**Technology & Tools**	**Risk Management**	**Estimation & Scheduling**	**Design**	**Process**
Clear Statement of Requirements [38][33]	x									
Clear Vision and Objectives [38]	x									
Recognize what is most important [47]	x		x							
Clear Project Mission [45]	x									
Use of specific requirements method [50]	x									x
Complete and accurate requirements from the start [50]	x									
Complete and accurate requirements received during the project [50]	x									
Enough allocated time for requirements elicitation [50]	x							x		
Project scope well defined [50]	x									
Executive Management Support [38][48]		x								

Ownership [38]		x								
Staff rewarded for long hours [50]		x								
Commitment & support from project sponsor/champion [50]		x								
Realistic Expectations [38][33]	x	x	x	x				x		
No scope creep [33][45]	x	x	x					x		
Proper Planning [38]			x					x		
Detailed project schedule [48]			x					x		
Good schedule estimations [50]			x					x		
Smaller Project Milestones [38]			x							
Ongoing evaluation by project manager [33]			x							
Project manager provide feedback [33]			x							
Experienced project manager [50]			x							
Risk identification before project start [50]			x				x			
Project manager understood customers problem [50]			x	x						
Customers & team have a good relationship [33]				x						
Customers provide feedback [33]				x						
Inclusive decision making [33]				x						
Co-located customers and teams [47]				x						
Good customers [47]				x						
Client acceptance [48]				x						
Sufficient communication [50]				x						
Customer involved in the project [50]				x						
Competent Staff [38][48]					x					
Hard Working, Focused Staff [38]					x					
Team sufficiently skilled [33]					x					
Sufficient access to business/technical experts [33]					x					
Small high performing team [33][45]					x					
Maintain continuity of personnel [47]					x					
Availability of required technology [48]					x	x				
Keep coding simple [47]						x				
Defensive coding principles [47]						x				
Good design [45]									x	
Defined development methodology [33]										x
Use an evolutionary life cycle and prototyping [47]										x

Table 3-2 Summary & Classification of Project Success Factors

Failure Factor	**Requirements, Objectives, Vision & Scope**	**Management & Sponsor Support**	**Project Management**	**Communication & Working Relationships**	**Knowledge, Skills Resourcing**	**Technology & Tools**	**Risk Management**	**Estimation & Scheduling**	**Change**
The project scope is ill-defined	x								
Incomplete Requirements & Specifications	x								
Unclear Objectives [55]	x								
Lack of User Input [38][44]	x			x					
Ambiguous business needs and unclear vision [44]	x								
Changing Requirements & Specifications [38]	x								x
Project managers don't understand users' needs [38]	x		x						
Business needs change [28]									x
Changes in design specification late in the project [44]									x
The chosen technology changes [28]						x			x
Project changes are managed poorly [28]			x						x
Project changed PM [50]			x						
PM supported long hours [50]			x						
Reactive and not proactive dealing with problems [44]			x				x		
Inadequate project risk analysis [44]			x				x		
Unmanaged Risks [54]							x		
Incorrect assumptions regarding risk analysis [44]			x				x		
Deadlines are unrealistic [28][38][44][45]		x	x					x	
Managers ignore best practices and lessons learned [28]		x							
Top down management style [44]		x							
Stakeholder politics [54]		x							
Inadequate or no management methodology [55]		x							
Commercial pressures [54]		x							
Bad planning and estimates [55]								x	

Consultant/vendor underestimate the project scope and complexity [44]								x	
Poor internal communication [44]				x					
Users are resistant [28]				x					
Extra staff added to meet deadline [50]					x				
High staff turnover [49]					x				
Sponsorship is lost [28]					x				
Lack of Executive Support [38]					x				
Unrealistic Expectations [38]					x			x	
Absence of an influential champion and change agent [44]					x				
Lack of Resources [38]					x				
Insufficient domain knowledge [55]					x				
Insufficient senior staff on the team [55]					x				
Poor performance by suppliers (hardware/software) [55]					x				
Technology Incompetence [38]					x	x			
Involve high degree of customization in application [44]						x			
Inappropriate choice of software [44]						x			
New or immature Technology [38] [55]						x			
Sloppy development practices [54]					x				

Table 3-3 Summary & Classification of Failure Factors

3.3 Success and Industry Type.

Berntsson-Scensson and Aurum [50] studied the different perspectives software practitioners have on the effect various factors have on project success among different industries. They focused on three industries namely the financial services industry, the consulting industry and the telecommunications industry. The study consisted of 27 responses, representing projects across the three different industries from companies based in Sweden and Australia.

Their first research question was: “What is the effect of certain factors on the success or failure of projects across various industries?” After a thorough literature study they selected 15 factors related to project success and failure. They then went further and asked the participants to indicate which of these factors characterized the projects that they participated in along with the outcome of the project (see Table 3-4). These results were then taken into account along with the outcomes of the projects to determine which factors related strongly to successful and failed projects respectively per industry. Their findings after analyzing the results are summarized in Table 3-5, which shows the percentage of successful and failed project respectively per industry where the factor was reported.

Factor	**Financial Services**		**Consulting Industry**		**Telecommunications**	
	Success (%)	Failed (%)	Success (%)	Failed (%)	Success (%)	Failed (%)
Project changed Project Manager	33	100	57	50	63	50
Project Manager supported long hours	50	50	33	100	100	100
Staff rewarded for long hours	17	0	50	100	50	50
Use of specific requirements method	100	100	100	100	100	0
Complete and accurate requirements from the start	17	0	29	0	29	0
Completed and accurate requirements during the project	100	100	100	100	67	0
Enough allocated time for requirements elicitation	67	0	71	0	50	0
Project's scope well defined	83	0	83	50	38	50
Extra personnel added to meet schedule timetable	33	100	29	100	71	100
Commitment and support from sponsor/project champion	100	100	71	100	100	50
Experienced project manager	90	70	40	50	80	60
Project manager understood the customer’s problem	80	90	70	60	70	40
Customer involved in the project	80	70	60	60	80	50
Risk identification before project start	70	40	80	60	80	50
Good schedule estimations	60	20	80	30	60	40

Table 3-4 Factors & Percentage of projects filling each factor Berntsson-Scensson [50]

Success Factors	**Failure Factors**
Financial Services (8 responses)	
Complete and accurate requirements from the start	Changing the project manager
Good schedule estimates	Adding extra personnel to meet schedule estimates
Enough time for requirements elicitation	
Well-defined project scope	
Reward staff for working long hours	
Consulting (9 responses)	
Complete and accurate requirements from the start	Project manager supporting long working hours

Enough time for requirements elicitation	Adding extra personnel to meet schedule estimates
Good schedule estimates	
Telecommunication Industry (10 responses)	
Complete and accurate requirements from the start	Inconclusive
Completing the requirements during the project if not available from the start	
Enough time for requirements elicitation	
Use of a specific method for requirements gathering	

Table 3-5 Success and failure factors by industry Berntsson-Scensson & Aurum [50]

They then identified 18 factors that were related to project success and asked the subjects to select the three most important and three least important factors for a project to be successful according to their beliefs. Table 3-6 shows these results.

	Financial Services	Consulting Industry	Telecommunications
Three most important success factors	Customer involvement	Very good project manager	Good relations between personnel
	Committed sponsor	Understanding customer's problem	Understanding customer's problem
	Overall good requirements	Well defined communication	Complete and accurate requirements
Three least important success factors	Good schedule	Good schedule	Good schedule
	Good relations between personnel	Committed sponsor	Experienced project manager
	Good estimates	Overall good requirements	Committed sponsor

Table 3-6 Software project success factors according to software practitioners Berntsson-Scensson & Aurum [50]

Comparing the importance of factors present on actual projects with the beliefs of practitioners presented some interesting findings. Berntsson-Scensson and Aurum noted that even though the practitioners from all industries did not rate "good schedule" as important, this factor was present on most of the successful projects evaluated. Subjects from the financial services industry also personally viewed "customer involvement" and "committed sponsor" as important. This was surprising as 70% of the failed projects in which they participated involved the customer, and 100% of the failed projects had commitment from the sponsor. When it comes to the consulting industry, the participants did not rate "understanding the customer's problem" as particularly important, which did not align with the characteristics of their last projects. Berntsson-Scensson and Aurum also reported a positive alignment between the beliefs of practitioners and the characteristic of their projects in the telecommunications industry when it came to "complete and accurate requirements" and "understanding the customers' problem".

3.4 Influential and Interesting Research Regarding Project Outcomes

3.4.1 GAO Study

Whenever a paper on the topic of software project failure was written in the 1980's, the most common source to cite was the GAO study. This was a study done by the U.S. Government Accounting Office to describe the horrific failure rates amongst studied projects. This study however quickly lost support after a researcher reread the paper and emphasized the fact that the study focused on projects that were known to be failing at the time of the study. In essence the study only proved that already failing projects tend to fail. [37]

3.4.2 Chaos Report

The Standish Group released the first version of the Chaos Report in 1994. The Chaos Report is a widely cited survey, but is not without criticism. The main objectives of the study were to determine the scope of project failures as well as the main contributing factors to project failure and key ingredients to reduce project failures. The first report in 1994 reported that 31.1% of projects are canceled before they are completed and a further 52.7% of projects are completed 187% over budget, with the remaining 16.2% completed successfully. [38] Subsequent Chaos reports showed subtle improvements in these figures. Section 2.3 elaborates on the Chaos report.

3.4.3 Addison Study

Addison and Vallabh conducted a study in South Africa in 2002 to identify the risks involved in a software project, rank them in order of importance and frequency of occurrence as well as identify the activities performed by project managers to control these risks. [56] The motivation for the study was the fact that 35% of abandoned projects were only terminated during the implementation phase of the life cycle, wasting vast amounts of resources. While conducting the study they identified 14 risks and 14 controls to manage these risks from literature. They then used questionnaires to see the importance and frequency of these applicable in practice with the following results.

Risk factors (in descending order of importance):

- Unclear or misunderstood scope/objectives
- Misunderstanding the requirements
- Failure to gain user involvement
- Lack of senior management commitment
- Developing the wrong software functions
- Unrealistic schedules and budgets
- Continuous requirement changes
- Inadequate knowledge/skills
- Lack of effective project management methodology
- Gold plating (see section 4.10 for definition)

Frequency of occurrence of risk factors (in descending order):

- Unrealistic schedules and budgets
- Continuous requirement changes
- Unclear or misunderstood scope/objectives
- Lack of senior management commitment
- Failure to gain user involvement
- Misunderstanding the requirements
- Gold plating
- Lack of effective project management methodology
- Inadequate knowledge/skills
- Developing the wrong software functions

When investigating which of the controls identified in literature were used to on the surveyed projects it was found that most of the controls were used most of the time. Exceptions were: developing of contingency plans to cope with staffing problems and not attempting too many functions on software projects.

Frequency of occurrence of controls (in descending order):

- Clearly assign responsibilities to team members
- Develop and adhere to a software development plan
- Management involvement during the entire project life cycle
- Dividing the project into controllable portions
- Stabilize requirements specification as soon as possible
- Involving users during the entire project life cycle
- Assess cost and schedule impact of each change to requirements and specifications
- Review progress to date
- Educate users on the impact of scope changed during the project
- Ensure there is a steering committee in place
- Combine internal evaluations and external reviews
- Include formal and periodic risk assessment
- Not attempting too many new functions on software projects
- Develop contingency plans to cope with staffing problems

3.4.4 KPMG Study of Runaway Projects

Glass used the findings of the 1994 KPMG survey in his book Software Runaways [35] to discuss some of the characteristics of so called “runaway projects” as well as typical courses of action taken on such projects. A runaway project was defined as a project that has “failed significantly to achieve its objectives and/or has exceeded its original budget by at least 30%. The survey was done across 120 organizations in the UK. It found that 62% of them had experienced at least one software runaway project. The main findings, causes and remedies undertaken are summarized in the following sections.

Main findings:

- Many of the runaway projects were overly ambitious.
- Most projects failed from a multiplicity of causes
- Management problems were more frequently the dominant cause than technical problems.
- Schedule overruns were more common than cost overruns.
- All sectors of software development are equally susceptible to failure.
- Use of packaged software did not help reduce the number of project runaways.
- Possible runaway projects were identified early on.
- New technology dramatically increased the chances of a runaway project.
- Risk management was rarely applied to runaway projects.

Causes of runaway projects:

- Project objectives not fully specified
- Bad planning and estimation
- Technology new to the organization
- Inadequate / no project management methodology
- Insufficient senior staff on the team
- Poor performance by Suppliers

Remedies attempted during runaways:

- Extending the time 85%
- Better project management procedures 54%
- More people 53%
- More funds 43%
- Pressure on suppliers by withholding payment 38%
- Reduction in project scope 28%
- New outside help 27%
- Better development methodologies 25%
- Pressure on suppliers by threat of litigation 20%
- Change of technology used on project 13%
- Abandon the project 9%
- Other 9%

3.5 Project Evolution Outcome

Various Project Evolution participants were interviewed to determine how they define project success or failure in general and what factors they believe typically contribute to the successful or failed outcome of a project. They were then also asked to rate the outcome of Project Evolution and identify areas where problems were experienced on the project.

3.5.1 Defining Success in General

Most of the project participants indicated that for a project to be seen as successful it needs to meet business requirements, within budget and on schedule.

The project manager placed great emphasis on the fact that it is the only the agreed upon functionality that needs to be delivered for the project to be judged as successful. Interestingly he stated that the requirements need not necessarily be the right requirements, but as long as they are agreed upon and delivered the project should be judged as successful when it reaches a delivery date. He expressed the view that the requirements are allowed to change and some conflicts might arise, but ultimately it is what is agreed upon which must be delivered. He made a clear distinction between a systems project and a business initiative, indicating that business users often want a system to solve all their problems, which he sees as unfair as often many of their problems lie in their business process, and should be addressed there. He feels that often the system or project is judged to be a failure when in fact it is the business process that is failing the business. He also noted that in his experience projects are generally never regarded as an absolute success or failure. When asked what role schedule and budget plays in the outcome of the project he replied that they are mainly used as measures for reward.

Apart from meeting business requirements on time and within budget, the architect also placed emphasis on the maintainability of the system over time. The reason he gave for this was the fact that he believes the largest project costs lie in maintaining the system over time. When asked to describe a failed project he described it as one which is not implemented or does not meet the users' requirements or has a sufficient level of quality or is grossly over budget.

The systems analyst once again felt that if a project is delivered with the requested functionality within, on time and within budget it can be seen a successful. She also stated that is possible to compromise on schedule, budget or scope, and still be successful, but it depends on the type and context of the project. She felt that generally failed or compromised projects are judged so because they fail to deliver on requirements.

Users generally placed greater importance on the delivery of required functionality as well as the accuracy and trustworthiness of the system. Developers generally felt that they succeeded when the solution was elegant and worked. They also felt that if they delivered what was asked from them and the system is being used it can be seen as a success.

3.5.2 Contributing Factors in General

It was striking that most factors participants believe to contribute to the successful or failed outcome of a project are requirements or project management related. It is important to note however that even though the participants were asked to give contributing factors in general, their answers were probably influenced by the most recent project in which they had participated on, this being Project Evolution.

The systems analyst believed that the following factors contribute to the successful outcome of software projects:
Agreement on requirements between stakeholders.
A good understanding of the requirements amongst all stakeholders.
A good project plan with clear responsibilities.
A controlled scope.
Good and timely communication, especially regarding the project status.

When asked about causes of failed projects she listed mostly the opposite reasons, including poor planning, bad quality requirements, lack of communication and misunderstandings regarding the scope.

In the architect's opinion the following factors typically contributed to project success:
Motivated and willing participation and the will to succeed.
A real business need and the will to succeed.
Communication and good project management.
Developers that know what they are building.
Tight scope management.
Risk mitigation.
Clear vision and objectives.

He felt that too large a focus in any one aspect of the project management triangle, namely scope, time or cost, would negatively affect project outcome. He also mentioned people and process troubles such as lack of business buy in, political motivations, lack of adequate resources and poor communication and decision making as contributing factors to challenged projects.

When it came to ensuring the successful outcome of projects, the project manager felt strongly that managing expectations amongst stakeholders is crucial. This required good communication. The developers on the other hand felt that good requirements together with a realistic schedule and proper testing are essential to the successful outcome of a project.

3.5.3 Project Evolution Outcome

When asked to comment on the outcome of Project Evolution, mixed responses were given. The project manager believed that although challenged, many aspects of the project can be seen as successful, and that although the business is struggling to adopt the new process and technology, they are now in possession of a better solution that will bring them many benefits in future. He also believed that although the system did not provide all the necessary functionality for the business to function fully by only making use of it in the beginning, they will be able to do so in future.

The project manager felt that there were two areas that jeopardized the project outcome most severely. The first was the late delivery and low quality of user requirements, and the second was the fact that expectations were not adequately managed. He believes that the project schedule needs to be divided percentage wise between the different activities that need to be performed, but because the requirements process took longer then expected and because the project sponsors were not willing to make any further allowances, the time allocation to the various remaining activities were skewed. It was also necessary to reduce the scope drastically in order to meet the first phase of delivery. The project manager also felt that there were many conflicting expectations across various levels in the organization and stakeholders, and that there were not communicated and managed successfully, inevitably leading to disappointment amongst certain groups.

Quoting the Systems General Manager from an email, dated 21 August 2008, sent to the systems team and executive management: "On the 1 June 2008 Group Risk implemented the 1st phase of the Evolution System – this was a major implementation that was done in a very short time in light that the business requirements were completed on the 20 December 2007. Unfortunately the business confidence in the implementation was very low. We therefore embarked on a project 'Business Confidence Testing' to identify defects and resolve before volumes were introduced into Evolution. I'm pleased to announce the results of this initiative which proves that the implementation of the Evolution Project was very successful and a major achievement in the Group Risk space." The email goes further to describe 17 defects that were identified, while 9 were rejected as "training issues", 8 were resolved in the 3 months post phase one deployment. It is important to note that the defects identified were only related to functionality that was included in the renegotiated scope for phase 1 and did not include the any of the missing functionality which was of importance to the business users.

The architect flatly described the project as a failure in general, stating lack of business buy in and bad technology choice as the main reasons for giving this response. This was an interesting and unexpected response from this participant. In an informal conversation with the architect early on in the project implementation, he stated that he believes very few projects can actually be seen as failures in practice. He believed that only projects that were outright canceled before they are completed could possibly be seen as failures and that most projects are in fact only challenged. Because of this apparent change of heart the architect was then asked to describe his feelings towards the project outcome at various stages before and after delivery. He replied that while implementing the project he thought of it as successful, directly after delivery he thought of the outcome as challenged because it did not provide the users with the necessary functionality and performance was poor. Three to six months after delivery he felt that it was still challenged due to it not being maintainable and still not able to deliver the required functionality. After six months he felt that it was severely challenged because it was still not able to meet requirements, and the system was still not being used at that stage. It is important to note that the architect resigned roughly six months after implementation. Most of the architect's responsibilities lay with the java team implementing the policy quoting system. When asked about his reasons for leaving he indicated that they included the fact that he did not believe implementing Compass as the policy administration system was the right decision, and that there were no challenges left for him as "the java work was mostly dumb". It is thus possible that his drastically different view on the project outcome might be due to the fact that he was only around for a short period after the initial delivery – a period during which the team arguably experienced most of the turmoil and dissatisfaction from the business users. His view on the outcome could also be negatively influenced by the negative or hostile feelings that typically prompt one to seek employment elsewhere.

The systems analyst felt positive about the overall outcome of the project. She felt that while they faced many challenges while implementing the system, the delivered system should be seen as a success as it met all its agreed upon requirements and was delivered on time. She did concede that shortly after the phase 1 release date the project could be described as compromised due to the fact that the reduction in scope in order to meet the deadline was not properly communicated amongst different business teams, and the perception amongst users were that the system did not deliver on its promises, and that it will not allow them to perform their duties. She reports that roughly 12 months after implementation the system is in full use and that the users seem happy to use it. When asked whether she believed that the new system is an improvement on they old system she replied that at this stage she thinks the two systems are roughly on par, but that the new system holds many future benefits. She believes that they are now better equipped to respond to changing business requirements in a shorter turnaround time than before and that the new system will be able to provide functionality not previously possible in the old.

When asked to identify problem areas on Project Evolution, she replied that the ever changing requirements coupled with poor communication both internal to the systems team and between the systems and business teams posed the biggest challenges. She also mentioned inadequate planning as a problem area.

Studying the project charter the following project success factors were formally specified:
Executive buy-in and support.
A shared vision between business & systems.
Timely access to business and subject matter experts.
Adherence to best practices, rigorous documentation and compliance to standards.

These factors are all very subjective in nature and the degree to which they were present on the project can be argued and certainly changed over time. The executive team certainly supported the project financially but later on enforced a strict unrealistic deadline. There were some clear divisions between business and systems and systems often voiced complaints that they were not receiving timely responses from business, delaying the process. A number of best practices and standards were set in place, but as time became limited these were then compromised, testing being one such example. At the inception of the project copious amounts of documentation in the form of requirements, project plans, score cards, risk evaluations, reports and technical documentation were generated. However these were rarely updated from mid-implementation onwards.

Table 3-7 comments on the various categories of factors contributing to project outcome as described in section 3.2.

Category	**Comment**
Requirements, project vision, objectives, scope	The project clearly experienced problems with late, low quality requirements. Requirements also changed frequently as deficiencies became evident. The project vision and objectives seemed to be clearly defined and communicated. The project did not experience too much scope creep per se, but additional necessary functionality was identified at times, mainly due to deficient requirements. The scope was also scaled back as time progressed to meet deadlines.
Management and sponsor support	The project certainly had management and sponsor support at inception, but it could be argued that it lost this somewhat to the end and after delivery. The project was however marked by politics and power struggles between business units.
Project management	The project had an experienced project manager who followed many project management best practices and produced large amounts of project management documentation. The participants did however report problems with project planning and scheduling and the management of expectations.
Communication and working relationships	The project fared poorly as far as communication and working relationships were concerned, especially between business and systems teams, but also between different systems teams.
Knowledge, skills and resources	The project struggled to find adequately skilled resources at inception, and struggled with a high degree of turnover throughout the project.
Technology and tools	The project made use of a large number of new technologies and tools unfamiliar to both the systems team and business team.
Risk Management	Studying the project documentation, risks were identified, documented and tracked along with mitigation strategies. Whether this was done proactively or reactively was not always clear.
Estimates and scheduling	The project schedule was arguably unrealistic. Most of the project activities started later then expected. Estimations were generally poor, and implementations often took longer than expected and it is speculated that they also cost more than expected.
Design	A number of design issues were identified, and maintenance became messy.
Process	At the onset of the project a decision was made to follow the RUP methodology. This was done to some degree, although the percentages of time spent on each phase and various activities were somewhat skewed.
Change	The system required many changes throughout the implementation and after delivery.
Testing	Testing time was limited. The testing team was also not sufficiently trained to use the system and minimal formal testing plans were available. No automated testing was done. Even though a testing tool was obtained, it was mainly used

	as a bug tracking system only. Overall testing coverage was poor.

Table 3-7 Project Evolution Success & Failure Factors Summary

3.5.4 Conclusion

Many similarities and differences were clearly noticeable between the different project participants' views on project outcomes. Firstly it was striking how all the participants defined project success as delivering on requirements within budget and schedule. It was also interesting that they did not mention customer satisfaction as requirement without being prompted to comment on it. Even though none of the participants mentioned that project outcome changes over time, they clearly indicated that it does by describing it as initially successful because they delivered on time, then as challenged 0 – 6 months after delivery, and then as either successful or failed 6 – 12 months after delivery.

When we evaluate the project outcome against popular definitions and criteria of project success found in literature, the project seems to fall within the grey area between successful and failed, or compromised. One criterion that it definitely met was on time delivery. This was arguably one of the key project requirements. The project team was markedly proud of this achievement, and one often got the sense that the Systems General Manager was proud of the fact that his team was working under extremely tight and arguably unrealistic timelines. One of the participants once commented that the Systems General Manager had a reputation in industry for getting the impossible things done and that this was why he was hired for the position.

As for delivering the required functionality, the project certainly delivered what was left in the agreed project scope after it was cut down in order to meet the deadline. But it needs to be said that the quality and performance of the delivered system left plenty to be desired. It also became evident that not all the stakeholders were made sufficiently aware of just how a limiting a set of functionality would be delivered. The users of the system were also not able to perform their duties fully due to the poor performance of the system and gaps in functionality. Several bugs were also found in the implemented functionality. This could possibly be attributed to the lack of end-to-end systems testing due to the implementation phase extending into the testing phase. The fact that the users refused to use the system at one stage and reverted back to the old system clearly indicates that the system failed as far as customer satisfaction was concerned. So the question remains, did the project truly deliver the required functionality?

Obtaining actual figures on the project budget proved difficult. It is not clear what the original project budget was or whether it was overshot or not. No actual figures could be seen in any of the project management documentation, although provision was made in the document templates to indicate projected and actual figures. One participant commented that the Systems General Manager was given unlimited budget as long as he delivered the system on time. This however seems unlikely. Two interesting observations were however made. Around three months prior to delivery, the developers were all instructed that overtime will be paid for any work done after hours and over weekends and that they need to do what ever needs to be done to reach their delivery dates as they will be compensated for it. This resulted in a vast amount of overtime logged which had the ability to derail even the most generous project budget. Shortly after delivery all contracts with contracting staff were terminated, and positions were left vacant when staff resigned. The reasons given for this was budget constraints. This could lead one to the conclusion that the project costs were more then intended.

Did the system meet its objectives? The business required a solution that will allow them to cut their operating costs, increase customer satisfaction, increase the amount of active policies administered on the system and allow them to implement new products faster. While the system is certainly more flexible and extendible in future and able to administer more business in future, in the short term it definitely impacted negatively on their business goals. This is mainly due to the slow performance of the system and missing functionality. The new business team was not able to prepare quotations in time, resulting in frustrated brokers and lost business. Due to gaps in functionality the administration teams were also not able to activate many new schemes, which were quoted for in the old system on the new administration system to be administered and generate income. This also caused delays in claim payments etc. Overall administration costs were driven up because of overtime that was necessary to clear backlogs, and customer satisfaction went down because of delays and errors.

From this case study we can clearly see that different stakeholders indeed have different success criteria, and that the outcome of a project does change over time. It is thus extremely necessary to manage expectations across the groups of stakeholders and ensure that the true criteria for success are identified to optimize satisfaction in the short, medium and long term. In this case I believe this was not done successfully. A skewed emphasis was placed on schedule, severely limiting scope up to a point where the system was made unusable in the short term. This left business in a vulnerable position. Even though the project was mostly judged to be successful by the systems team, the main business of the organization is not writing software but selling life insurance, and for this reason I feel that a greater importance should have been placed on what is good for the business. In hindsight emphasis on scope rather then schedule could have led to a better outcome in the short and medium term while the long term benefits of the new system would still have been in place. Overall the project cannot be described as a success as it clearly did not meet all its shareholders expectations and was in fact unusable for a certain period of time. But it can also not be described as a failure as it is capable of delivering on requirements and objectives in the long term and was never canceled. It thus falls in the grey area where most projects fall, namely challenged.

When it came to problem areas present on Project Evolution, the participants unanimously agreed that they faced many challenges due to poor and changing requirements as well as unmanaged expectations and poor communication. It is for this reason that we choose to further investigate requirements engineering in the following chapter. In chapter 5 we will then investigate architectural design and architecture reviews in particular. The reason for this is as follows. Firstly performance was one area in which the system experienced many problems. These were mainly due to architectural design choices. Secondly a system's architecture serves as a means of communication amongst stakeholders, another area of concern.

Performing architecture reviews also allows for requirements to be further clarified and non-functional requirements such as performance to be explicitly specified and the architecture design to be evaluated against these. Getting the requirements and architecture right early on in the project is said to limit the amount of change and rework necessary later on during implementation, and are thus areas worthy of focus.

4 Requirements Engineering

Part of every software development team's goal is to develop software that meets the users' requirements. It became clear in the previous chapter that requirements play a critical role in the final outcome of the project as it was often stated as a reason for project success or failure. In this chapter we will investigate various aspects surrounding software requirements, recommended practices and their impact on the project outcome.

The outline for this chapter is as follows:

Firstly we will start off by looking at the importance of good requirements in section 4.1. We will then attempt to find a definition of requirements engineering in section 4.2. Once we understand what is meant by requirements engineering we will identify some of the challenges typically encountered during the requirements engineering process in section 4.3. In sections 4.4 and 4.5 we will identify the types of requirements to consider and the stakeholders from whom they should be obtained. Once we know what we are looking for and where to find them, sections 4.6, 4.7 and 4.8 elaborate on some of the processes involved in requirements engineering, namely, requirements elicitation, requirements analysis as well as aspects involved in specifying and documenting requirements. Once requirements are documented it is then time to assess their quality and this is discussed in section 4.9. Merely having quality requirements only partly fulfills the goals of good requirements management. Prioritizing them and managing the scope of the project forms another integral part of requirements management - we will discuss an approach proposed by Leffingwell and Widrig in section 4.10. We will also investigate aspects of requirements engineering in compliance frameworks such as CMM, ISO 9000 and REGPG as well as development methodologies such as RUP and XP. Finally we will discuss the requirements engineering process followed on Project Evolution.

4.1 The Importance of Good Requirements

"Requirement deficiencies are the prime source of project failures." – Glass' law [52]

It's important to firstly define what is meant by a requirement in the context of software development. Various definitions of a requirement exist. Two popular ones include:

Dorfman and Thayer define a requirement as [5]:
"A software capability needed by the user to solve a problem or to achieve an objective. This software capability must be met or possessed by a system component to satisfy a contract, standard, specification or other formally imposed documentation."

The IEEE standard 279 defines it as [7]:
"(1) a condition or capability needed by a user to solve a problem or achieve an objective; (2) a condition or capability that must be met or possessed by a system to satisfy a contract, standard, specification or other formally imposed document."

A quote from Brooks [42] illustrates why requirements are so important: "The hardest part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all of the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."

"Errors are most frequent during the requirements and design activities and are more expensive the later they are removed" – Boehm's first law [52]

Many studies indicate that poor requirement specification and requirements management are to blame for software projects failing or being compromised. It has been shown that errors occurring during the requirements phase of a project can have a negative effect on not only the usefulness of the system to the users but also on the project budget and schedule as well as on team morale.

Davis [7] provides a number of hypotheses to this effect. Firstly Davis says that the later in the development lifecycle an error is detected, the more expensive it will be to repair. This is supported by independent studies done in the early 1970's by GTE, TRW and IBM all reaching roughly the same conclusion that the cost to detect and repair an error during the maintenance phase was twenty times as much as during the requirements phase. This is illustrated by Figure 4-1 while Figure 4-2 illustrates the cumulative effects of errors.

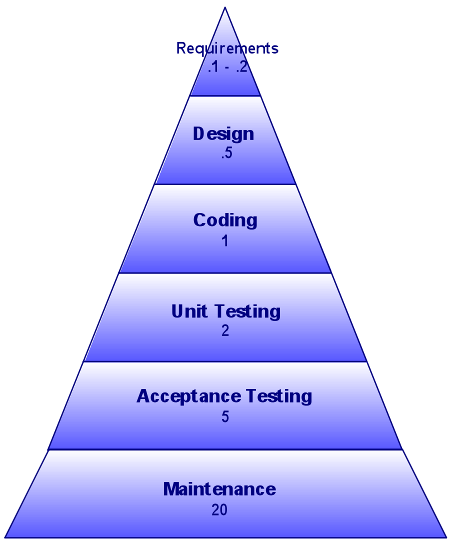


Figure 4-1 Relative cost to repair a defect at different lifecycle phases [7]

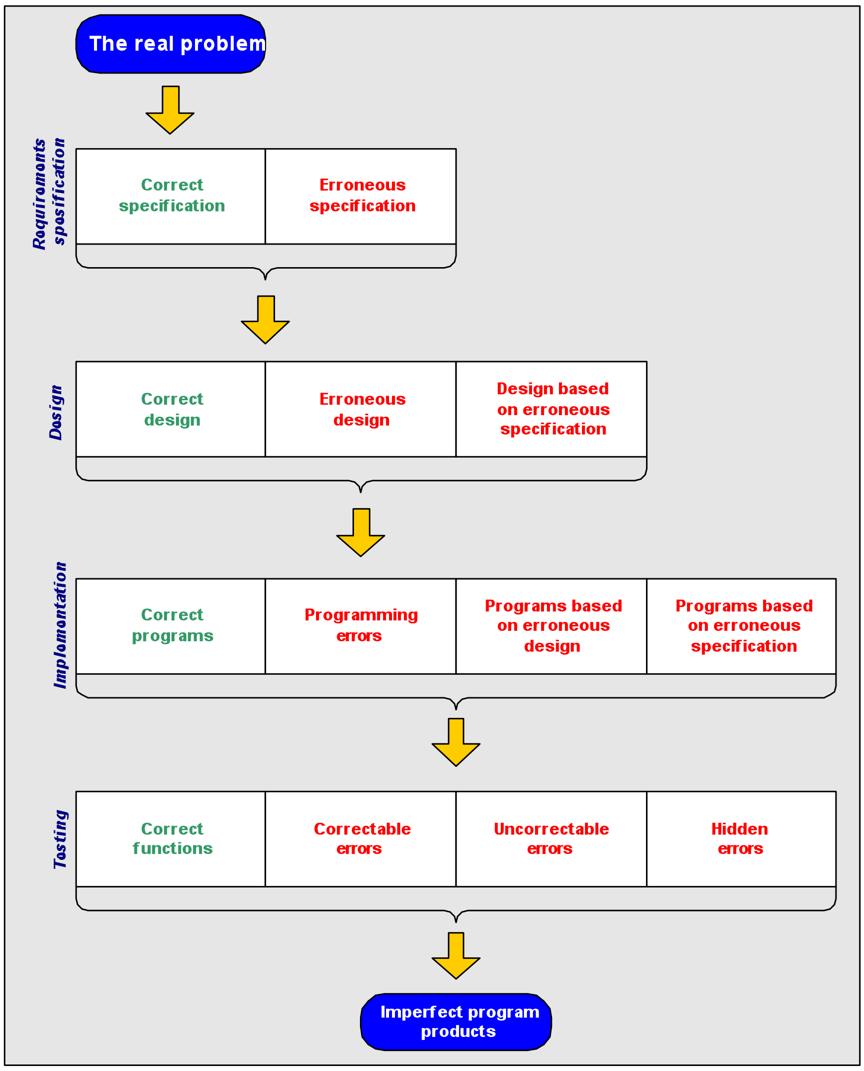


Figure 4-2 Cumulative Effects of Error [7]

Davis states that many errors remain latent and are not detected until well after the stage at which they were made. This is supported by a conclusion made by Boehm that 54% of all errors ever detected in software projects studied at the TRW organization were detected after the coding and unit testing stages, and that 45% of those were attributed to the requirements and design phases rather than the coding stages. Many errors are also being made during the requirements phase of the project. Davis stated that 56% of all bugs detected can be traced back to the requirements phase [7].

Davis [7] puts forward that errors made in requirement specifications are typically incorrect facts, omissions, inconsistencies and ambiguities. He supports this statement by quoting figures from a study by the Naval Research Laboratory on the operational flight program of the Navy A-7E aircraft. The data showed that 77% of all requirements errors were non-clerical.

Also, according to Leffingwell & Widrig [5], requirements errors top the number of delivered defects, contributing to approximately a third of all defects. They too agree that it is much more expensive to fix a requirement error in later stages of the project costing between 50 and 100 times more, depending on its nature and where in the SDLC it is found, than fixing it during the requirements phase of the SDLC.

The reason for requirement errors being so costly to fix is due to the increasing amount of work that needs to be redone the later they are found in the SDLC. A requirements error found during unit testing could require changes affecting the design and coding that was completed before. Typically requirement errors are also hard to spot after the requirements phase of the project because the team tends to focus on design issues during the design phase, and coding errors during coding etc because they believe the requirements to be complete and correct. It is also considerably more effort to verify a requirements error during later stages of the life cycle because the user, business analyst or systems analyst that provided the requirements would need to be consulted. They may not be readily available or at this stage might have forgotten the original requirements causing delays in rectifying the defect.

Repairing defects related to requirement errors are likely to incur costs in the following areas: re-specification, redesign, recoding, retesting, change control, data fixes and damage control, scrapping of completed code, recall of defective versions, warranty costs, product liability, service costs and documentation.

Damian et al. [78] discovered the following benefits to conducting requirements engineering while undertaking a case study on the topic:

- Improved understanding of details, dependencies and complexities of features.
- Reduced wasted effort.
- Improved communication.
- Enabled informed decisions.
- Contributed to better estimations.
- Provides a mechanism for controlling change.
- Increased productivity.

Given the frequency of requirements errors occurring and the cost to fix them, and that they can easily consume 25% - 40% of the project budget, it is thus fairly easy to see why starting the project off on the right foot during the requirements phase is very important to contribute to the successful overall outcome of the project. Doing so has been shown to have a direct impact on the quality, cost and timely delivery of the system.

4.2 Requirements Engineering Defined

The terms requirements management and requirements engineering are often used interchangeably in the literature, and definitions often vary by author. In some instances requirements management forms a sub-process of requirements engineering.

Leffingwell and Widrig define requirements management as [5]:
"A systematic approach to eliciting, organizing and documenting the requirements of the system and a process that establishes and maintains agreement between the customer and the project team on the changing requirements of the system."

Westfall defines software requirements engineering as [6]:
"A disciplined process-oriented approach to the definition, documentation and maintenance of software requirements throughout the software development life cycle."

Westfall [6] further divides requirements engineering into two major processes namely requirements development and requirements management. Requirements development encompasses all the activities involved in eliciting, analyzing, specifying and validating the requirements. Requirements management according to Westfall commences after the baseline requirements have been agreed upon and includes activities such as requirement change control, impact analysis, approving of baseline changes and implementation of such changes. This is illustrated by Figure 4-3.

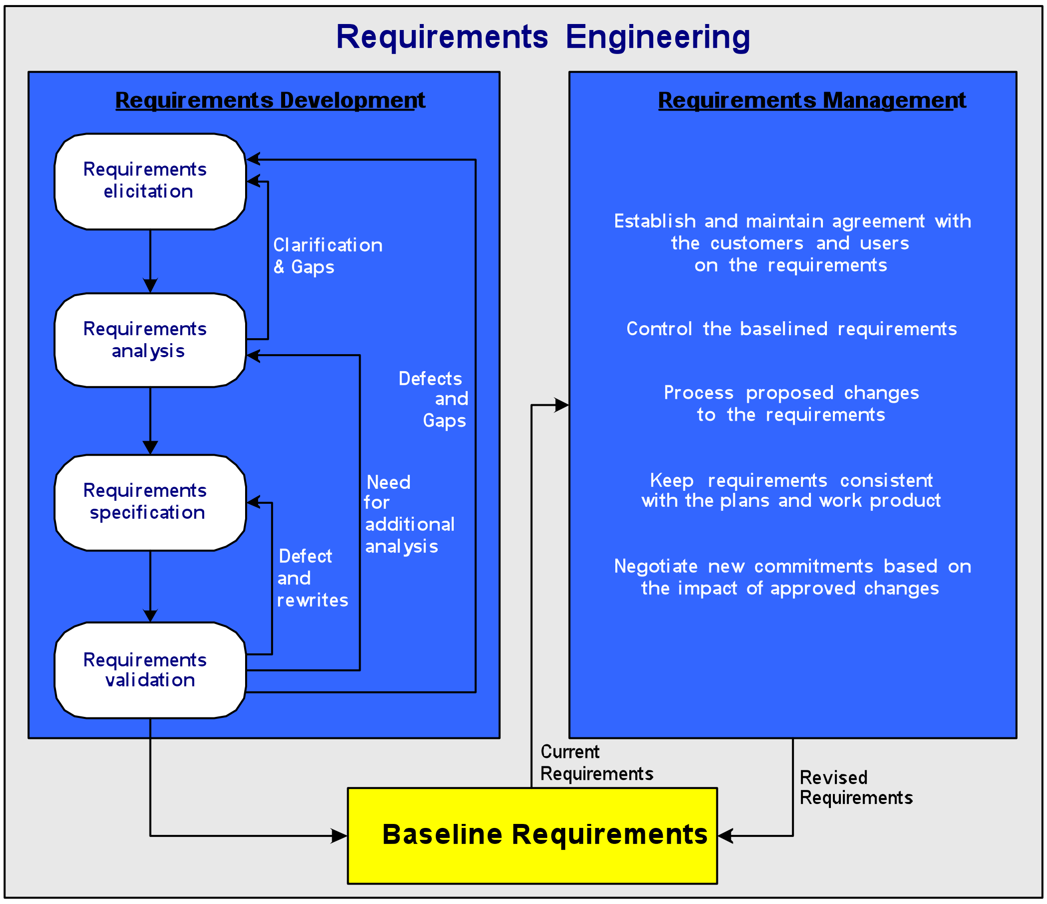

Requirements Engineering
Requirements Development
Requirements elicitation
Requirements analysis
Requirements specification
Requirements validation
Clarification & Gaps
Defects and Gaps
Need for additional analysis
Defect and rewrites
Requirements Management
Establish and maintain agreement with the customers and users on the requirements
Control the baselined requirements
Process proposed changes to the requirements
Keep requirements consistent with the plans and work product
Negotiate new commitments based on the impact of approved changes
Current Requirements
Revised Requirements
Baseline Requirements

Figure 4-3 Requirements engineering phases & activities [6]

Although various definitions exist for requirements engineering, all practicing requirements engineers have five primary purposes: (1) determining what needs exist, (2) determining what needs to address, (3) selecting the appropriate solution, (4) documenting the intended external behaviour of the system to be offered, and (5) managing the ongoing evolution of the needs and the solution's intended behaviour. [70]

4.3 Challenges to Requirements Engineering

A lot of research has been conducted in the area of requirements engineering as well as on the challenges faced during requirements engineering. Many common challenges have been identified across different industries. This section will summarize some of the conclusions drawn by researchers in the field.

One study in particular that is cited by many is that of Curtis et al. [59] published in 1988 entitled "A field study of the software design process for large systems". In their article they identify 3 prominent problems often encountered during systems development. They are: firstly, the thin spread of application domain knowledge; secondly, fluctuating and conflicting requirements; and lastly, communication and coordination breakdowns.

Lubars et al. [58] conducted a field study of 10 organizations to discover how they define, interpret, analyze and use requirements for their software systems and products. They studied both market-driven and customer-specific projects. The following points were mentioned by practitioners as sources of difficulty during the requirements engineering process.

- Vaguely stated or missing requirements.
- Requirement misunderstandings and faulty assumptions.
- Lack of customer contact.
- Frequently changing requirements.
- Difficulty negotiating and prioritizing requirements.
- Lengthy, overly complex or insufficient requirements documentation.
- Difficulty expressing or mapping requirements to models as well as understanding of models by non-specialist users.
- Difficulty modeling system behaviour.
- Difficulty relating performance requirements to specific parts of the data- or control flow.

Building on the work done by Curtis et al. [59] and Lubars et al. [58] is that of Emam et al. [60]. In their research they collected data on 60 projects and focused only on the requirements engineering process. They concluded that in addition to technical issues, non-technical issues are of major importance during the requirements engineering process in terms of problems, solutions and impediments to implementing the solutions. One issue of importance is the level of detail that goes into the functional requirements or model. Too much detail as is said to often be delivered by inexperienced analysts is not cost effective, while too little detail, as is often the result of a lack of resources, does not provide adequate information to the development team. Another area of concern is examining the existing systems of the organization. It is possible to waste resources when examining the existing systems in too much detail, but sometimes organizations are also reluctant to spend any further resources on existing systems. Emam et al. recommend that existing systems be studied up to a degree where it is possible to identify their short-comings as well as any components that might be reused. Analyzing existing systems is also useful for educating junior analysts about the problem domain. User participation is another area of requirements engineering that proved to be problematic. Although user participation is crucial it is said to sometimes be hampered by the relevant users not be available. It may also be hampered by a lack of cooperation or open hostility between the IS department and users. Other areas of concern during the requirements engineering process upon which Emam et al. elaborate, include managing uncertainty and project management capability.

Kamsties et al. reported additional challenges faced during the requirements engineering process when surveying requirement engineering practices in small and medium sized enterprises [61]. They found that often there was a lack of documented requirements. In the cases where requirements were documented they were often incomplete and imprecise or, at the other end of the scale, too overly complex to understand. Often the requirements were too vague, rendering them un-testable by anyone but the domain experts. Lastly requirements were often said to be untraceable. This meant that new requirements or changes to existing requirements would affect virtually all the components of the system already implemented.

Karlsson et al. conducted a study focusing on the challenges faced during the requirements engineering process specifically on market-driven software projects [57]. They defined market-driven products as products vended on the open market to a large range of potential customers, thus requiring a diverse set of requirements to be considered. While some of the challenges they identified are very specific to the market-driven software development domain, others are clearly relevant to software development in general. The 12 challenges they identified are as follows (Note that the challenges that Karlsson et al. consider special to market-driven organizations have been marked with an asterisk.):

- Lack of simple techniques for basic needs.
- Communication gap between marketing staff and developers. *
- Writing understandable requirements. *
- Managing the constant flow of requirements.
- Requirements volatility.
- Requirements traceability and interdependencies.
- Requirements are invented rather than discovered. *
- Implementing and improving requirements engineering within the organization.
- Resource allocation to requirements engineering.
- Organization stability.
- Selecting the right process.
- Release planning based on uncertain estimates. *

Hofmann and Lehner [62] once again list lack of traceability and rapidly fluctuating requirements as significant challenges in requirements engineering. They also noted that often there is a lack of total system requirement definition that includes performance, capacity and external interface requirements. Other problems encountered include, unrealistic requirements by marketing, weak project management practices and difficulty prioritizing and communicating requirement priorities to stakeholders.

Many common challenges to requirements engineering were identified in these various studies. Changing requirements and lack of resources or knowledge are two such commonalities reported by most studies. Another common problem identified is that of documentation either being overly complex of not sufficient. Lastly a number of problems relating to the management process and communication were also frequently mentioned.

4.4 Requirement Types and Levels

When starting out with the requirements engineering process it is important to realize that requirements can be classified both by what they describe as well as by where they come from. Westfall [6] defined the following levels and types of requirements as an adaptation of those defined by Karl Wiegers (2004). They are aimed at creating awareness amongst practitioners in order for them to recognize the information they need to elicit, analyze, specify and verify when defining software requirements. A number of requirement types exist on a business-, user- and product level as illustrated in Figure 4-4.

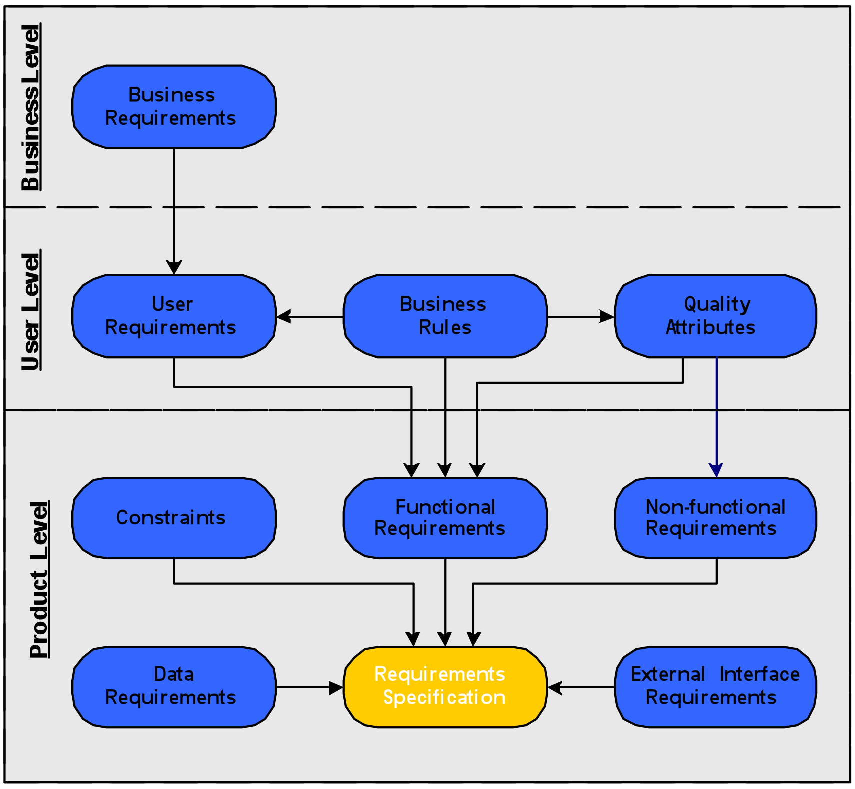


Figure 4-4 Levels and Types of Requirements

4.4.1.1 Business Level Requirements

Business level requirements define the problems that the business wants to solve or opportunities that they hope the system will address. These requirements mainly define why the system is being developed. They contain the objectives of the requesting organization or group.

4.4.1.2 User Level Requirements

User level requirements focus on what the users of the system require to achieve by the business objectives defined in the business requirements. The functionality needed by the users of the system is captured in the user requirements. There will typically be multiple user requirements for each business requirement stated.

Business rules are policies, procedures and guidelines that dictate how the users need to perform tasks to achieve the business objectives and are therefore seen as user level requirements. The software needs to adhere to and implement these rules.

Lastly, the quality specifications to which the system should to conform on a user level are defined in the quality attributes.

4.4.1.3 Product Level Requirements

Functional requirements specify the software functionality that is necessary to achieve the user requirements. Once again a single user requirement may result in multiple functional requirements.

Some quality attributes defined on the user level requirements may be translated into functional requirements on the product level as specific functionality that has to be implemented to meet the requirement. In other instances quality attributes could be translated into non-functional requirements if they merely specify characteristics that the system should have to meet the required level of quality.

If the system needs to interface to external systems, all requirements for interfacing to external entities need to be included in the external interface requirements.

The constraints delimit restrictions that might be imposed on the system that could affect the design of the solution.

The data requirements specify specific data items or data structures that must be included in the product.

4.5 Stakeholders Involved in Requirements Engineering

Westfall [6] defines stakeholders as individuals who affect or are affected by the software product. They therefore have some level of influence over the requirements for the software product. It stands to reason that identifying the various stakeholders involved and their needs is an important step in specifying complete and accurate requirements. The stakeholders provide the analyst with an opportunity to access a wide base of experience and domain knowledge.

Westfall [6] identifies 3 categories of stakeholders, the acquirers of software, the suppliers of software and other stakeholders. Acquirers are the customers who requested the software and/or who pay for it. Suppliers include everyone involved with the development and distribution of the software. There are also other stakeholders who might not be directly involved with the system, but who may be indirectly affected by it or have an interest in it. Various stakeholders under these three categories can be found in Table 4-1.

Acquirers	**Suppliers**	**Other**
Customers End-users	Requirements Analysts (Business and Systems Analysts) Designers Developers Testers Documentation writers Project managers	Legal / Contract management Product release management Sales & Marketing Upper management Government / Regulatory agencies Society

Table 4-1 Stakeholder Categories and Types

Having access to the stakeholders provides the analyst with the opportunity to analyze, synthesize and resolve conflicts in input from the stakeholders and ultimately, to produce a complete set of requirements without having to use-as Wiegers (2003) puts it-the most ineffective requirement elicitation techniques, namely clairvoyance and telepathy.

Clearly input from the various stakeholders is very valuable to the requirements engineering process, but they first need to be identified. Westfall [6] provides a checklist of questions, as seen in Table 4-2, that can be used to identify stakeholders.

- What types of people will use the software product?
- What business activities are supported by the software product, and who performs, is involved in, or manages those activities?
- Whose job will be impacted by the introduction o the new software product?
- Who will receive reports, outputs or other information from the software product?
- Who will pay for the software product?
- Who will select the software product or its supplier?
- If the software product fails who will be impacted?
- Who will be involved in developing, supporting and maintaining the software product?
- Who knows about the hardware, other software, or databases that interface with this software product?
- Who establishes the laws, regulations or standards governing the business activities supported by the software product?
- Who should be kept from using the software product or from using certain functions/data in the software?

• Whom does the software product solve problems for?
• Whom does the software product create problems for?
• Who does not want the software product to be successful?

Table 4-2 Stakeholder identifying questions

Clearly having identified an extensive list of stakeholders brings forth the possibility of conflicting or contradictory needs. The next logical step would be to prioritize stakeholders based on their contribution towards the success of the project in order to provide a basis on which to evaluate their needs. Westfall [6] refers to a stakeholder-inclusion strategy that can be used to determine if one wishes to be friendly, unfriendly or ignore the stakeholder needs. Being friendly to a stakeholder entails considering and accommodating the needs of the stakeholder. If the stakeholders' needs are not integral to the success of the project, they may be ignored. Alternatively the analyst might consider being unfriendly to a particular stakeholder by counteracting her needs as they might pose a threat to the system.

Another matter that needs to be considered is who is to represent the different groups of stakeholders. Choices include representatives, sample groups, or exhaustive groups. In the case of homogeneous groups of stakeholders, it would make sense to choose a representative to stand for them. For large stakeholder groups a sample of individuals might be chosen to represent the group. If the stakeholder group is very small or essential to the success of the project, the entire group might be included.

Once the participating stakeholders have been chosen it is necessary to decide at what times during the process their presence and input is needed. How they will be involved is also important, be it through questionnaires, interviews, workshops, documentation reviews etc.

4.6 Requirements Elicitation

Now that we have identified the kinds of requirements we need to take into consideration as well as the stakeholders from where we are to source them, we are ready to start eliciting the requirements.

4.6.1 Barriers to Requirement Elicitation

It is common for the users to be disappointed when they see the first demonstration of a new system. They often react with "yes but…" This is frequently the result of insufficient, misstated or misconstrued requirements. Therefore it is important to be aware of certain barriers to requirements elicitation.

One such a barrier is the very *nature of software systems [5]*. Software systems are very much intangible until they are very close to the end product. This means that the user is often only able to indicate if the end product is indeed what he envisioned after much investment on the developer's part was made.

Systems people often remark that users don't know what they want. It is important for developers to keep in mind that users are not always able to articulate what they want or need to do. Furthermore each requirement found will in turn necessitate more requirements. A statement by a user: "We want to be able to do xyz" is often followed by: "Ok, but what if…" from the developer. Typically the user would not have thought that particular scenario through and will be unable to provide an immediate satisfactory answer, which will be frustrating to the developer. An even worse outcome of this newly discovered requirement could be changes required to other already discovered requirements. This is referred to as the "*Undiscovered Ruins*" syndrome by Leffingwell and Widrig *[5]*.

Another reason why it is so difficult to obtain a common understanding of the problem and the desired solution is the divergent worlds of developers and users. There is a huge communication gap between technical and business people because they typically come from different backgrounds, have different motivations and objectives and often even speak different languages. Leffingwell and Widrig *[5]* refer to this phenomenon as the "*User and the Developer*" Syndrome.

4.6.2 Requirement Elicitation Techniques

Since teams are rarely given complete requirements, they need to go out and get them. There are various techniques described by Leffingwell et al. [5] that can be used for discovering user requirements for a system and ensure a common understanding amongst users and developers. They include:

- Interviewing & questionnaires
- Requirements workshops
- Brainstorming and idea reduction
- Storyboards
- Use Cases
- Role playing
- Prototyping

-

4.6.2.1 Interviewing & Questionnaires

Interviewing is applicable to most situations. It is however important to note that this requires a face to face conversation between business and technical people. The "User and the Developer" syndrome should be thus be kept in mind, the biases and predispositions of the interviewer should not interfere with the free exchange of information and understanding of the problem.

It is not recommended to use questionnaires as a substitute to interviews, one reason being the inability to decided on all the appropriate questions beforehand, unlike an interview where the interviewer is able to adapt the questions asked based on the answers given. It is also difficult to clarify responses given to answers afterwards. Questionnaires are often perceived as being efficient, but their restrictive nature limits their usefulness. Questionnaires are however valuable to determine preference for a given limited subset of choices.

4.6.2.2 *Requirements Workshops*

The requirements workshop is the single most powerful requirements elicitation technique that can be applied in many circumstances.

A requirements workshop typically runs for 1 to 2 days attended by key stakeholders of the project and facilitated by either an experienced team member or, better yet, by an outside facilitator. The goal of a requirements workshop is to rapidly gain consensus on requirements and agreement on the course of action that should be taken to meet these requirements amongst stakeholders.

Requirement workshops have the benefit of gathering together all the stakeholders so they are all allowed to have their say. It can expose and possibly solve certain political issues that could jeopardize the outcome of the project and build an effective, committed team. An immediate preliminary system definition is reached along with a subsequent agreement between stakeholders and the development team.

4.6.2.3 *Brainstorming and Idea Reduction*

Brainstorming helps to find undiscovered requirements and aids completeness. Brainstorming also has other benefits such as participation; it encourages out of the box thinking and results in a broad set of possible solutions in a relative short amount of time.

4.6.2.4 *Storyboarding*

Storyboarding uses a number of diagrams and drawings in sequence to illustrate the intended working of the system. Storyboarding is used to show the who, what and how of a system and user behavior. It helps elicit early “yes but” responses as well as aid the discovery of further requirements. With storyboarding the users’ reactions can be observed well before any development is done and possibly even before any specifications have been written. Storyboards are recommended in all projects where the early gaining the of the users’ reactions is a key success factor. Other benefits to storyboarding include the fact that it is extremely inexpensive, user friendly, informal and interactive. It provides an early review of the user interfaces of the system and is easy to create and modify.

Storyboards are very helpful in situations where users are unsure of what they want as a storyboard will in the very least elicit a “no that’s not what we want” response. Storyboarding speeds up the conceptual development of the application as well as understanding and visualization of the business rules involved.

4.6.2.5 *Use Cases*

Use cases describe the interactions between users and the system, focusing on what the system does for the user. Use cases are defined as describing a sequence of actions a system performs that yields a result to a particular actor. Use cases play a significant role in various phases of the SDLC.

Once the notion of a use case is explained to the user it provides a simple structured language between the users and developers to work on requirements together. Once all the use cases, actors and objectives of the system are identified, each functional requirement can be explored further in the detailed functional behaviour. It is however important to note that use cases focus primarily on functional requirements and are not very helpful in identifying non-functional requirements.

4.6.2.6 Role Playing

During role playing the developer simply plays the role of the user for a couple of hours. Typically the developer will sit with a user and perform his or her function for a while. It's worth noting that role playing is not suitable for all scenarios.

Role playing provides the developers with the opportunity to experience the system from the users perspective, as observation alone may not be enough to gain a true understanding of the problem they are trying to solve. It often happens that users have particular workarounds due to certain real problems; these problems might not become clear to the developer who is simply observing the way the user is working. Many users may also not admit to these alternate procedures that they follow when questioned. It is impossible for the developer to know all the questions that they should be asking.

Role playing is another quick and cheap way for the developer to a gain real understanding of the requirements. Due to the nature of role playing it is very likely that developer will remember the insights gained by the experience and even develop an empathy for the user. It does however require developers to move out of their comfort zones and interact with people.

4.6.2.7 Prototyping

Prototypes allow the user to interact and experience the system with a limited subset of functionality early on in a way that none of the other elicitation techniques provide. This has obvious benefits such as getting out the majority of "yes but" responses as well as eliciting further requirements.

Prototypes can serve as a confirmation from the user that the developer understands the requirements and it can also serve as a catalyst to encourage further requirements.

4.6.2.8 Documentation Studies

Westfall [6] suggest that studying documentation related to the industry, organization or current systems provide useful insight into the problem domain and can be seen as another requirements elicitation technique.

Documents that may be included in the study are:

- Industry standards, laws or regulations
- Product literature
- Process documentation and work instructions

- Change requests
- Prior project reviews
- Reports and other deliverables from current systems.

-

4.7 Requirements Analysis

Once the requirements have been elicited it is necessary to check them for consistency, completeness and feasibility. Any arising conflicts then need to be resolved through prioritization and negotiation with the stakeholders. A renegotiated set of requirements is then agreed upon. This process is what is referred to as requirements analysis [67]. Other than a list of prioritized agreed upon requirements, another output of requirements analysis is often a set of models that can be used to bridge the gap between analysis and design later on. These could include data-flow models, semantic data models or object-oriented models.

One of the main techniques used during requirements analysis is Joint Application Development or JAD sessions for short [67]. During these sessions developers and customers discuss the desired features of the system to be developed. The purpose of these sessions is to define the project at various levels of detail, to design a solution and to monitor the project till completion. JAD promotes cooperation, understanding and teamwork amongst stakeholders from varying backgrounds.

4.8 Specifying and Documenting Requirements

Since not all stakeholders can nor should be involved in every part of the requirements engineering process, it is necessary to document the requirements in some form in order to communicate them amongst the various stakeholders and participants. Well managed projects often adhere to the principal that requirements only exist if they are documented. [52]

When Lethbridge et al. [64] researched software engineers' attitudes towards documentation, they found that even though documentation is often outdated it still provided some useful information, or at least historical guidance for maintainers. Table 4-3 summarizes the respondents' beliefs on the usefulness of software documentation when completing certain tasks. They also found that systems often have too much documentation and that much of the mandated documentation is so time consuming to create that its cost outweighs its benefits. Documentation is also often poorly written and finding useful content often proves to be such a challenge that people might not even try to do so. This paints a pretty bleak picture for software documentation in general, but the fact remains that requirements need to be communicated between stakeholders in some way that can be referenced when needed. In this section we will look at some of the considerations surrounding documenting system requirements.

<u>Task</u>	<u>Percent</u>
Learning a software system	61
Testing a software system	58

Working with a new software system	54
Solving problems when other developers are unavailable to answer questions	50
Looking for big-picture information about a software system	46
Maintaining a software system	35
Answering questions about a system for management or customers	33
Looking for in-depth information about a software system	32
Working with an established software system	32

Table 4-3 Percentage of survey respondents who rated documentation effective or extremely effective for particular tasks [64].

4.8.1 Documenting Platforms

When choosing a platform to document requirements, according to Decker et al. [63], it is important to choose one that can support effective, efficient collaboration among a large number of diverse stakeholders. They also list a number of challenges the chosen platform should account for concerning the stakeholders. Firstly it should account for different perspectives on the system under development. Secondly it should cater for the different stakeholder backgrounds which can cause communication problems. Thirdly it should take the different objectives which influence the views on the requirements into account. It should also provide for the different abilities to express requirements and document them using a technical platform and, lastly cater for the different stakeholder involvements, such as only certain stakeholders are allowed to make decisions.

Decker et al. [63] describes 3 main categories of tools commonly used to document requirements. Firstly there are office suites such as MS Office or Open Office or even plain text documents. Then they listed general collaboration tools such as MS Sharepoint or Google's Writley and lastly they listed dedicated requirements engineering tools such as Telelogic's Doors or Borlands CaliberRM. Table 4-4 describes these tools in more detail and lists some of the pros and cons of each.

	Plain office suite/text documents	**General collaborative tools**	**Dedicated RE tools**
Description	· Requirements are stored in office software suites and word processors.	· Requirements are stored in collaborative tools	· Requirements are stored in dedicated RE tools
Pros	· Support for structuring requirements in sections · Low cost and wide spread availability · Also applicable in other	· Support for collaboration · Support for grouping requirements into individual documents · Support for structuring	· Support for collaboration · Support for grouping requirements into individual documents · Support for structuring

	development phases such as testing	requirements into sections · Also applicable in other development phases	requirements into sections · Support for versioning and baselining of requirements
Cons	· Collaboration chaos from concurrent changes by different stakeholders or late changes via email · Distribution chaos if requirements are distributed across several documents; high risk that document links will break · Un-typed links (link semantics can't be captured) · No explicit versioning and baselining of requirements	· Distribution chaos if requirements are distributed across several documents; high risk that document links will break · Untyped links (link semantics can't be captured) · No explicit versioning and baselining of requirements	· Specialized tools that aren't optimized for non technical users. · High cost, especially for small and medium-sized enterprises that might need a license for every stakeholder.

Table 4-4 Pros and cons of tools commonly used in requirements engineering [63]

Another specific tool or platform that Decker et al. [63] suggest be used to document requirements is a Wiki site. They believe wiki's to be powerful yet lightweight tools that are well suited to documenting ever changing requirements. Some of the benefits they believe can be gained from using wiki's during requirements engineering include: Support for collaboration; Support for requirements versioning; Low cost; No tool or licenses required; Applicable to other development phases. The use of wiki's for requirements is however not without out drawbacks. It is for example difficult to export page content and users must remember page names to be linked. There is also no explicit baselining of requirements and it is difficult to version content across a number of pages. Wiki's are also missing replication of wiki content (for working offline).

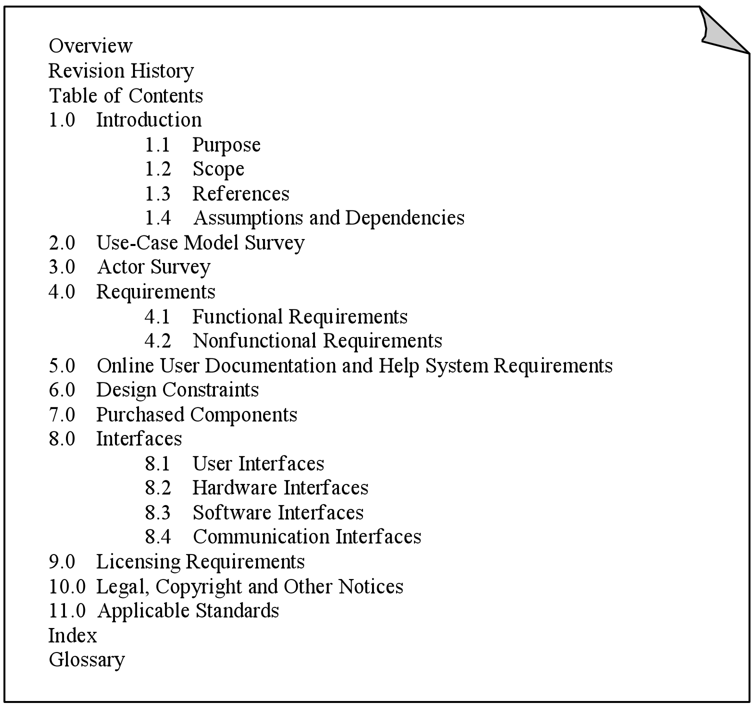
4.8.2 The System Requirement Specification (SRS)

Leffingwell et al. [5] describes the modern SRS as a logical structure rather than a physical document. The projects vision document serves as the main input to the SRS documents. The SRS is an elaboration of various requirements for the system to be built embodied in a package. It describes the complete external behaviour of the system. It is usually the developer team members or systems analyst that takes ownership of the SRS package.

The SRS plays a number of important roles [5]:

- It serves as a basis for communication among all parties.
- It serves as an agreement amongst various parties on what functionality they are accountable to deliver.
- It serves as a software manager's reference standard for discussions with the project team.
- It serves as input to the design and implementation teams.
- It serves as input to the quality assurance and testing teams.
- It controls the evolution of the system throughout the project.

During the lifespan of the project the SRS documents will primarily be used as reference items. It is thus important that they are organized in a format that familiar and easy to understand and navigate. When organizations are looking to qualify for higher levels on the SEI-CMM scale or be classified as ISO 9000 compliant more formal formats are generally required. The Figure 4-5 describes one possible format of a SRS document [5].



Overview
Revision History
Table of Contents
1.0 Introduction
    1.1 Purpose
    1.2 Scope
    1.3 References
    1.4 Assumptions and Dependencies
2.0 Use-Case Model Survey
3.0 Actor Survey
4.0 Requirements
    4.1 Functional Requirements
    4.2 Nonfunctional Requirements
5.0 Online User Documentation and Help System Requirements
6.0 Design Constraints
7.0 Purchased Components
8.0 Interfaces
    8.1 User Interfaces
    8.2 Hardware Interfaces
    8.3 Software Interfaces
    8.4 Communication Interfaces
9.0 Licensing Requirements
10.0 Legal, Copyright and Other Notices
11.0 Applicable Standards
Index
Glossary

Figure 4-5 SRS Document example layout

4.8.3 Technical Methods for Specifying Requirements

Leffingwell et al. [5] suggest augmenting natural language specifications with requirements specified using technical methods when the requirement description becomes too complex for natural language, or if one cannot afford to have the requirement misunderstood. This will be the case for systems dealing with life-and-death situations or where erroneous behaviour could have extreme financial or legal consequences.

There is a variety of technical specification methods available. Although a full discussion of these methods is beyond the scope of this dissertation, the following list provides an indication of some of the better known methods:

- Pseudocode
- Finite state machines
- Decision trees
- Activity Diagrams
- Entity Relationship models
- Object-oriented analysis
- Structured analysis

4.8.4 Media-rich Methods of Capturing Requirements

Due to the availability of digital devices such as digital cameras, voice recorders and video recorders and the greater accessibility to bandwidth and the internet, media-rich content creation is becoming an inexpensive and everyday activity in requirements engineering according to Gotel and Morris [65]. Even though using media-rich content in capturing requirements seems like an easy, quick and comprehensive method of requirements gathering, Gotel and Morris warn that traceability is easily and often lost using this technique. According to them a traceability process usually involves an engineer creating links between fragments of natural-language text, supposedly interrelated to show each requirement's development path, from its original source material to the eventual system components. Traceability aids requirements understanding, helps manage requirement changes and demonstrates requirements satisfaction. Gotel and Morris recommends that when insisting on the use of content-rich media in projects, it is important to establish an approach that distinguishes between media fragments that are worth keeping ready-to-hand and those that can be archived just for a full record. They also recommend being selective when creating media records, as media incurs costs when organizing for traceability.

4.8.5 Documentation Standards

Although most organizations determine an in-house standard for their requirement documentation, a number of industry standards exist to guide the documenting of requirements. One such standard is IEEE/ANSI 830-1984 also known as the Guide for Software Requirements. Figure 4-6 shows the general overview of the standard, but it is important to note that the standard recognizes the fact that different types of systems need different emphasis in their requirements. The standard thus offers four different alternatives to organize specific requirements. The different alternatives take into account that for some systems, it might be easier to describe all external interfaces in terms of their use, or that non-functional requirements might not be global and could be better defined together with their particular functional requirements.

1. Introduction
1.1 Purpose of SRS
1.2 Scope of product
1.3 Definitions, acronyms and abbreviations
1.4 References
1.5 Overview of rest of SRS
2. General description
2.1 Product perspectives
2.2 Product functions
2.3 User characteristics
2.4 General constraints
2.5 Assumptions and dependencies
3. Specific requirements
Appendices
Index

Figure 4-6 IEEE/ANSI 830-1984 Overall Outline

4.8.6 Maintenance

When specifying requirements, especially when using technical methods, it is important to keep maintenance in mind. By using technical methods sparingly and only to illustrate the behaviour of the system, the amount of maintenance required can be reduced considerably [5]. Outdated documentation could be a major pitfall, since it might be misleading. This is why many developers believe in letting the code be the documentation. If time does not permit documents to be updated accurately, one alternative could be to simply label them as outdated as not to mislead anyone. As Leffingwell put it, having a stale model is often better than not having any model at all, as long as you know it is stale. Another manner to alleviate the problem of maintaining technical requirement documentation is the use of new software development tools that support round trip engineering, thus keeping requirement documentation and code in sync. Lethbridge et al. [64] determined by means of a survey that when software engineers do update documentation, it is often only done weeks later as can be seen in their results in Figure 4-7. They also found that software engineers tend to use and update simple yet powerful documents more regularly as opposed to lengthy complicated documents such are requirement documents or specifications. The simple methods for documenting included bug-tracking systems, test cases or inline code comments.

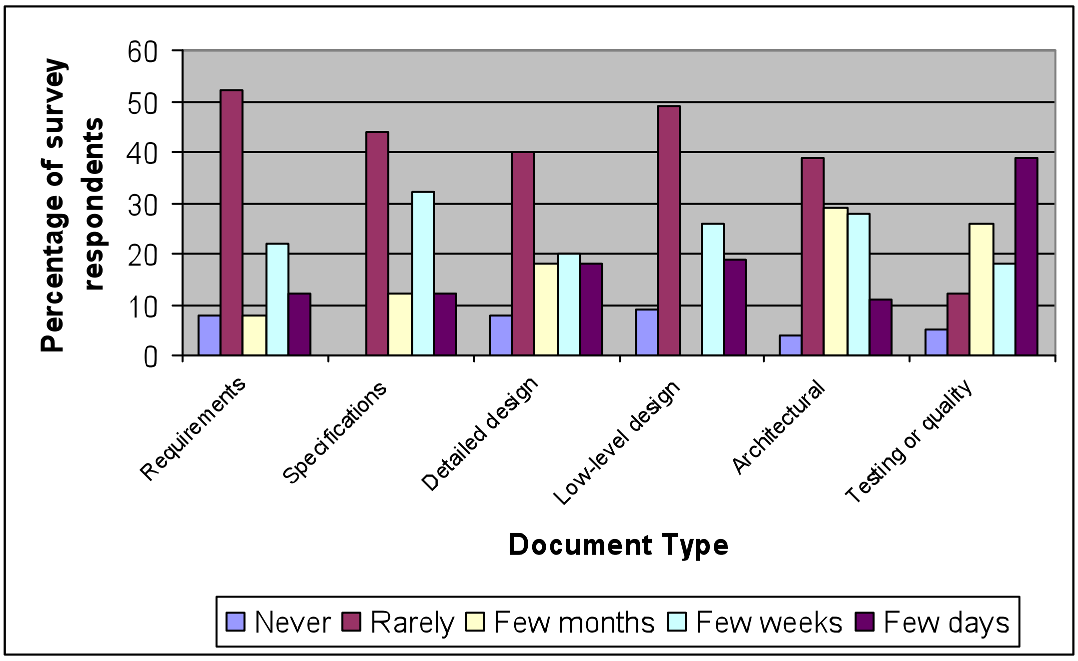


Figure 4-7 The time between system changes and document changes for various document types. [64]

4.9 Quality Measures and Validation for Software Requirements

Software developers cannot build what cannot be properly specified. The quality of the requirements specified has a direct bearing on the quality of the software. Measuring the quality of the requirements is thus an important part of requirements engineering. However, as Singh and Sabharwal puts it, software requirements are intellectual entities: finding out of measurable attributes, measuring them and also having standards to compare them with is a challenging task [66].

When considering quality during the requirement engineering process, according to Leffingwell and Widrig [5] it is important to consider the quality of the individual requirements, the set of requirements as well as the SRS package as a whole.

The following quality criteria should be considered when evaluating individual requirements [5][7]:

- *Correctness*

  Every requirement needs to represent something that is required of the system to be built, additional requirements that are not needed should be omitted.
- *Unambiguous*

  Each requirement should only have one possible interpretation and be easy to understand.
- *Concise*

  Requirements should be short and specific

- *Finite*
  Requirements should not be stated in an open-ended manner and should not contain words such as "all" or "throughout".
- *Measurable*
  When applicable requirements should always have quantifiable limits or values associated with them.
- *Feasible*
  It should be possible to implement the requirement using available resources and technology.
- *Testable*
  It has to be possible to determine if the system satisfies the given requirement.
- *Traceable*
  Each requirement should be traceable back to its original source and this source should be easily reference-able in the requirement.
- *Understandable*
  Both the user and developer communities need to fully comprehend the functionality implied by the requirement.

When evaluating the requirements set one should take the following into account [5]:

- *Completeness*
  A set of requirements is only said to be complete if it describes all significant requirements of concern to the user including requirements associated with functionality, performance, design constraints, attributes or external interfaces. It should also define the appropriate response to valid and invalid inputs. It should also provide references and labels for all the figures, tables and diagrams and should clearly define all the terms and units of measure used.
- *Consistent*
  The individual requirements within the requirements set should not be in conflict with each other. The level of detail should also be consistent across individual requirements.
- *Requirements ranked for importance and stability*
  It is important to know the users perception of what is important or likely to change when determining and managing the scope of the project. High quality requirements should thus include this information.
- *Modifiable*
  The structure of the requirements set should be such that changes can be made easily, completely and consistently while retaining the existing structure and style of the set.

As already stated, it is also important to evaluate the SRS package for quality. Leffingwell and Widrig [5] concluded that when evaluating the SRS package, the most important criteria to consider is the ease with which information can be found. They mention 4 ways to facilitate access to information and thus improve the quality of the SRS package. Firstly the SRS should have a good table of contents, secondly it should include a good index, it should have a revision history and lastly a glossary.

There are a number of methods that can be used to evaluate the quality of requirements. If the requirements are very formally or technically defined and documented, certain tools exist that can evaluate certain quality aspects of the requirements. In practice most requirements are however stated in natural language. One method for assessing the quality of the requirements that is applicable to requirements stated in all forms is simply to evaluate them against a checklist of criteria. Another method that can be used even when dealing with requirements specified in natural language, is to conduct peer reviews of requirements. Writing test cases for requirements before the actual design and development starts is another good method that can be used to validate requirements. A functional set of test cases that only requires knowledge of the functional requirements and not of the internal working of the product will help identify missing or defective requirements.[5]

4.10 Requirements Management (Managing Scope)

Being on time and within budget are two of our primary objectives when it comes to building successful systems. One of the biggest reasons for failing to do so is simply biting off more than one can chew. It is important to be realistic about what is possible in a given time period and to manage the project scope accordingly.

Project scope is a function of the functionality or features that must be delivered, the resources available to do so and the timeline in which to achieve it. This is illustrated in Figure 4-8.

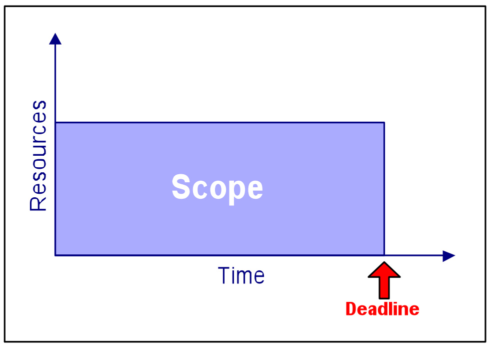


Figure 4-8 Project Scope

In practice most projects undertaken are over scoped by up to 200% [5]. Typically these systems will either be shipped incomplete or the deadline will be grossly missed, and they will almost always be of a shocking quality because quality assurance and testing will most probably have been reduced to a minimum or obviated completely.

The first step in successfully managing project scope according to Leffingwell and Widrig [5] is to compile a requirements baseline. This is an itemized set of features or requirements intended to be delivered in a specific version of the application. There are two important requisites for the requirements baseline, it needs to be at least acceptable to the users and it needs to be reasonable to the developers. According to Leffingwell and Widrig there needs to be between 25 and 99 features on the list. Having less than this might not be sufficient to estimate the amount of effort necessary to achieve the desired outcome and having more will make it difficult to communicate effectively between the customer and developer.

The next step will be having the users prioritize the selected features. It is important that this be done independently of any input from the technical community, as this might influence the usefulness of the end product. This should depict what is important to the users.

Once the features have been prioritized an estimate of the effort to deliver them must be made. Even though it is extremely difficult to make an accurate estimate at this stage, it is crucial because the goal is to determine what is feasible to implement in this release and no time should be wasted elaborating on requirements that might not even be included in the scope.

The last step that needs to be taken before the initial scope is decided on is adding a risk estimate to each feature. The risk associated with each feature is the probability that the implementation of that feature might lead to adverse effects on the budget or schedule. This risk is determined by the development team. Once all the features have been evaluated in terms of risk, risk mitigation strategies need to be considered for all high and medium risk features.

It is now time to determine what features can be added to the current release. If the team is able to give a rough labour-based estimate, the baseline could be determined by adding the labour related to each feature until the deadline is reached. If however these estimates are not available, it becomes more of a gut-feel game. If the feeling is that the project is over scoped by 200%, the baseline of work to be done needs to be cut roughly be half. If the feeling is that this is still not achievable only the critical features should be considered for the initial release. If the features were correctly prioritized this should cut the baseline by 2/3.

ID	Feature	Priority	Effort	Risk	Comments
3	Feature 3	Critical	Med	Med	
5	Feature 5	Critical	High	High	
7	Feature 7	Important	low	Low	
V1.0 Mandatory Baseline: Everything above must be included or release will be delayed					
1	Feature 1	Important	High	Med	
2	Feature 2	Useful	Med	High	
V1.0 Optional: Do as many of the preceding as possible					
4	Feature 4	Useful	Med	Med	
6	Feature 6	Useful	Med	Low	

Table 4-5 Example V1.0 Baseline

It is important to engage the customer in the scope reducing exercise. If they feel part of the process then they are more likely to accept the resulting set of features [5]. The customers need to realize that it is in their best interest to have a practicable baseline. It is also important that they are fully aware of what will and will not be delivered in the current release so that they do not over commit themselves on features that will not be available.

Competent customers will always be demanding, but left unchecked they could jeopardize the success of the project. The focus should remain on enough functionality at the right time to meet the customer's real needs. This process will call on the negotiation skills of the project manager. The guiding principle in establishing the baseline should be to under-promise and over-deliver.

As soon as a baseline is determined, the development team needs to work on elaborating on the requirements further so that a more accurate estimation can be made. This could lead to either further scope reduction or possibly additional features being added to the scope if it is discovered that more time and resources are available than initially estimated.

Once a workable baseline is established it can be used to realistically track the progress of the project. This allows for reacting in a timely fashion to possible slippages and can be used as part of the change management process.

Requirements Churn & Gold Plating

Westfall [6] refers to requirements churn as changes made to requirements after the baseline was determined. A certain degree of requirements churn is to be expected to refine the developers' understanding of the system. However excessive requirements churn will occur if there were requirements missing from the original specification or if requirements were ambiguous, or if the specification was poorly written. Good requirement engineering practices will help avoid these kinds of requirement churn.

It often happens that functionality is added to the system that was not requested by the users. Developers often believe that they are adding value that the users will simply love. Westfall [6] refers to this as gold plating. It is also possible to have gold plating on the user's side when they request functionality that is not essential or necessary. Gold plating wastes resources by developing functionality that is not necessarily needed, and could cause unnecessary errors and unreliable code. It can be prevented by following a strict requirements engineering process whereby nothing is added to the baseline without proper consideration and analysis. Gold plating on the users' part should be eliminated by proper prioritization of requirements.

4.11 Requirements Engineering and Compliance Frameworks

A number of formal software development standards and compliance frameworks exist against which organizations can have themselves evaluated and certified. The majority of organizations that obtain these certifications do so for economic reasons as certain levels of compliance might be required to compete for lucrative contracts. Certain cost savings and/or schedule improvements have also been reported by organizations implementing CMMI process improvements [74]. Three formal certifications for which organizations can apply include the Capability Maturity Model, ISO 9000 and the ISO/IEC 15504 standard. Sections 4.11.1, 4.11.2 and 4.11.3 discuss these in more detail, specifically relating to requirements engineering. Section 4.11.4 discusses an informal model, REGPG, which can also be used to evaluate and improve the maturity of an organizations requirements process.

4.11.1 CMM and CMMI

In 1986, the Software Engineering Institute (SEI), at Carnigie-Mellon University began work on a process maturity framework to help software professionals improve the software development process. In 1993 version 1.1 of the Capability Maturity Model for Software (CMM-SW) was released. It defines 5 levels of software maturity for an organization and provides a framework to move from one level to the next. These levels along with their associated key process areas can be seen in Table 4-6. The CMM-SW guides developers through activities designed to help the organization improve its software process, with the goal of achieving repeatability, controllability and measurability. [5]

Level	Key Process Areas
1. **Initial:** Ad hoc, even chaotic; success depends solely on individual heroics and efforts.	Not applicable.
2. **Repeatable:** Basic project management to track application functionality, cost and schedule.	Requirements Management Software project planning Software project tracking and oversight Software subcontract management Software quality assurance Software configuration management
3. **Defined:** The process for management and engineering is documented, standardized and integrated. All projects use an approved, tailored version of the process.	Organization process focus Organization process definition Training program Integrated software management Software product engineering Intergroup coordination Peer reviews
4. **Managed:** Detailed measures of the software process and software quality metrics are collected. Both process and software products are understood and controlled.	Quantitative process management Software quality management
5. **Optimizing:** Continuous process	Defect prevention

improvement is enabled by use of metrics and from piloting innovative ideas and technologies.	Technology change management Process change management

Table 4-6 Levels of CMM-SW with key process areas [5]

The “key process areas” described in the CMM framework are activities that have been found to be influential in various aspects of the software development process and result in better software quality. From Table 4-6 it can be seen that the first key performance area to be addressed to move from level 1 to 2 is requirements management. Requirements management is summarized by the CMM as follows: “The purpose of requirements management is to establish a common understanding between the customer and the software team of the customer’s requirements.”

The software requirements specification serves as a central project document. It is required to be kept consistent with other project plans and activities. Software requirements are required to be documented, reviewed or analyzed, verified and managed. Although all of these activities are required by the CMM in order for an organization to improve its rating, specific methods or practices for doing so are not prescribed.

Since 1991 various Capability Maturity Models (CMM’s) have been developed for various disciplines such as software development (as described in previous paragraphs), workforce management and development, integrated product and process development. Having multiple models proved problematic to many organizations who wanted to focus their improvements efforts across multiple disciplines within their organizations. This was due to differences between their models as well as additional costs related to training, appraisals and improvement activities. It is for this reason that the Capability Maturity Model Integration (CMMI) project was formed to sort out the problem of having multiple CMM’s. The CMMI project team’s mission was to combine 3 specific models into a single improvement framework for organizations pursuing enterprise-wide process improvement: the Capability Maturity Model for Software (CMM-SW), Electronic Industries Alliance Interim Standard (EIA-SECM) and the Integrated Product Development Capability Maturity Model (IPD-CMM). This resulted in the CMMI for Systems Engineering and Software Engineering ver.1.1 (CMMI-SE/SW) [72]. In 2006 CMMI for Development ver. 1.2 (CMMI-Dev) was released. This model was the designated successor of the 3 source models (CMM-SW, IPD-CMM, EIA-SECM) all having been retired. The history of CMM’s is illustrated in Figure 4-9. The SEI is currently working on CMMI ver. 1.3.

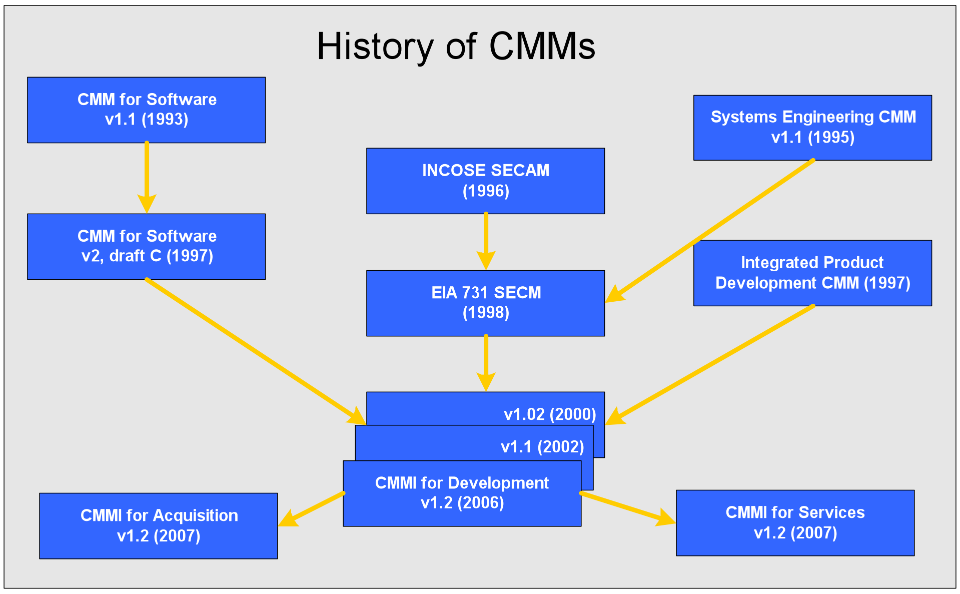

History of CMMs
CMM for Software v1.1 (1993)
CMM for Software v2, draft C (1997)
INCOSE SECAM (1996)
EIA 731 SECM (1998)
Systems Engineering CMM v1.1 (1995)
Integrated Product Development CMM (1997)
v1.02 (2000)
v1.1 (2002)
CMMI for Development v1.2 (2006)
CMMI for Acquisition v1.2 (2007)
CMMI for Services v1.2 (2007)

Figure 4-9 History of CMM's [73]

A detailed discussion on the CMMI-SE/SW and CMMI-DEV falls beyond the scope of this chapter. It is however important to note the role that requirements engineering plays in CMMI level progression. Recall that in the original CMM-SW model, requirements management was a required key process are to obtain level 2 maturity. In both the CMMI-SE/SW and CMMI-Dev models requirements management is still a key process area needed to obtain level 2, but to obtain level 3 maturity organizations also need to perform requirements development. At first glance it seems odd for organizations to be required to manage requirements before they are required to develop them, as stated by Linscomb [71]. It is when comparing the purposes described in the process areas that it becomes clear that on level 2 maturity the requirements need not be developed by the organization itself. Instead the requirements management process's purpose is solely to manage the requirements of the project's products and product components and to identify inconsistencies between those requirements and the project's plans and work products. In contrast the stated purpose of requirements development is to produce and analyze customer, product and product component requirements. [73]

4.11.2 ISO 9000

ISO 9000 consists of a number of quality management standards. The European Community has adopted ISO 9000 (naming it EN29000), and often requires ISO 9000 certification from companies wishing to do business in Europe.

Part 5.3 of the ISO 9001 document stipulates "In order to proceed with development, the supplier should have a complete, unambiguous set of functional requirements." It also states that the information should include all performance, safety, reliability, security and privacy requirements that collectively determine whether the delivered system is acceptable. The standard further calls specifically for [5]:

- Assignment of people from both the customer and developing organizations to be responsible for establishing requirements.
- Establishment of methods and procedures for agreement and approval of changes to requirements.
- Efforts to prevent misunderstandings of the requirements.
- Establishments of procedures for recording and reviewing the results of discussions about the requirements.

Once again the standard prescribes no specific methods to achieve these requirements.

4.11.3 ISO/IEC 15504

ISO/IEC 15504, also known as SPICE is yet another international standard on software process assessment. It was jointly developed between ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission). The architecture of ISO/IEC 15504 is two dimensional, as shown in Figure 4-10. The one dimension consists of the processes actually assessed. These are grouped into 5 categories. The second dimension consists of the capability scale that is used to evaluate the process capability [75]. The details of this standard falls beyond the scope of this discussion, but it is important to note that software requirements analysis forms one of the processes that can be evaluated, and falls within the Engineering process category.

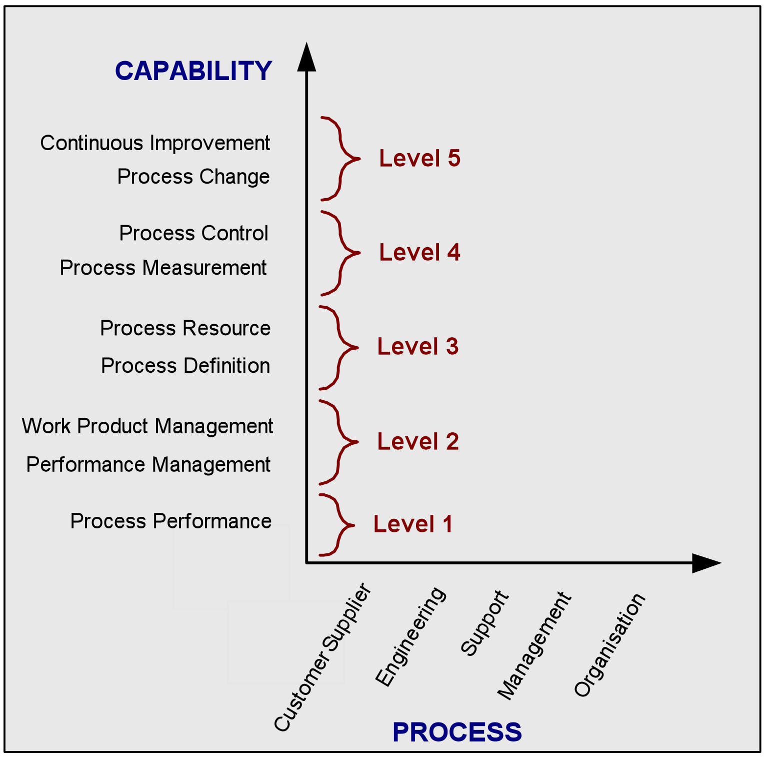


Figure 4-10 ISO/IEC 15505 Overview [75]

4.11.4 REGPG

The Requirements Engineering Good Practice Guide (REGPG) is an improvement framework developed by Saweyr et al. [77]. It draws on existing software process improvement models, like the CMM, to define a process improvement framework for systematic, incremental adoption of good requirements practice. REGPG is not used for formal certification, but was rather intended as a means for organizations to improve their requirements practices in order to gain cost and other benefits related to quality requirements.

REGPG classifies the requirements process being evaluated into one of three maturity levels, initial, repeatable and defined. It uses a checklist of 66 best practices to determine into which level of maturity the process can be classified. The best practices are classified as basic, intermediate or advanced. REGPG also provides an indication of the cost involved in introducing and applying the practice as well as the key benefits that can be gained through its adoption to further help organizations improve their requirements process. Basic practices represent fundamental measures that underpin a repeatable process. They are seldom technical solutions, but usually focus on defining organizational procedures or documentation standards. Intermediate practices are more technical and require most of the basic practices to be in place. Advanced practices require substantial specialist expertise and support continuous improvement. Table 4-8 contains a number of the best practices proposed by Saweyr et al.

The organization receives a score on each practice between 0 and 3. To receive a score of 3 (standardized), the practice needs to be a documented standard that is followed and checked as part of a quality management process. When the practice is widely followed but not mandatory, it is scored as 2 (normal use). If some project managers introduced the practice but it is not universally used, it is scored as 1 (discretionary). Finally if it is rarely applied it receives 0 (never). Table 4-7 shows REGPG maturity levels and assessment scores that are required to obtain each level. [77]

Level	**Description**	**Assessment Score**
1. **Initial:** Ad hoc requirements process	Organization finds it hard to estimate and control costs as requirements have to be reworked and customers report poor satisfaction. The processes are not supported by planning and review procedures or documentation standards but are dependant on the skills and experience of the individuals who enact them.	Less than 55 in the basic guidelines.
2. **Repeatable:** Defined standards for requirement	Organizations may use tools and methods. Their documents are	Above 55 in the basic guidelines but less than 40 in the

documents and have policies and procedures in place for requirements management.	more likely to be consistently high in quality and produced on schedule.	intermediate and advanced guidelines.
3. Defined: Defined process model based on good practices in place	The organization has an active improvement program in place and can make objective assessments of the value of new methods and techniques.	More than 85 in the basic guidelines and more then 40 in the intermediate and advanced guidelines.

Table 4-7 REGPG maturity levels and assessment scores

Good Practice	**Cost of Introduction**	**Cost of application**	**Classification**	**Key Benefit**
Uniquely identify each requirement	Very low	Very Low	Basic	Provides unambiguous references to specific requirements
Define policies for requirements management	Moderate	Low	Basic	Provides guidelines for all involved in requirements management
Define traceability policies	Moderate	Moderate to high	Basic to intermediate	Maintains consistent, traceable information
Maintain traceability manual	Low	Moderate to high	Basic	Records all project-specific traceability information
Use database to manage requirements	Moderate to high	Moderate	Intermediate	Makes it easier to manage large amounts of requirements
Define change management policies	Moderate to high	Low to moderate	Intermediate	Provides a framework for systematically assessing change
Identify global system requirements	Low	Low	Intermediate	Finds requirements likely to be most expensive to change
Identify volatile requirements	Low	Low	Advanced	Simplifies requirement change management
Record rejected requirements	Low	Low	Advanced	Saves re-analysis when rejected requirements are proposed again

Table 4-8 Examples of REGPG Good Practices [77]

4.12 Requirements Engineering in Different Development Methodologies

Requirements engineering is often seen as a traditional process, or a step in older waterfall or iterative type development methodologies. The reason for this is the heavy emphasis on upfront effort and reliance on documentation for knowledge sharing [67]. The standard waterfall life cycle model can be described as a sequential progression from requirements to design to coding to testing and finally to operations [7]. A popular iterative development methodology is the Rational Unified Process (RUP). Requirements engineering forms a workflow process within RUP and is mostly focused upon in the inception and elaboration phases as shown in Figure 4-11.

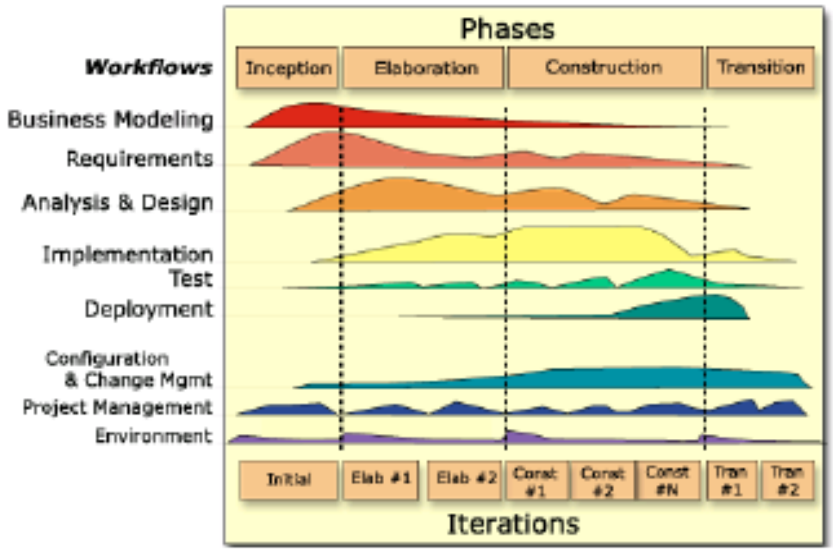


Figure 4-11 Overview of RUP [76]

More recently, agile approaches such as Extreme Programming (XP), Feature Driven Development (FDD) and Scrum have gained much popularity. The common principals among these agile methodologies include: improve customer satisfaction, adapting to changing requirements, frequently delivering working software and close collaboration between business and the development team [67]. Teams following an agile methodology are typically small in size. System requirements are stated in the form of features or stories, which are prioritized and developed over short development iterations, typically 1 to 3 weeks. Requirements are clarified by frequent interaction between developers and customers as they are implemented, and verified by test cases and prototype demonstrations. The systems design and architecture is usually a product of joint development team effort.

Although arguably both traditional and agile approaches have the same final objectives, requirements engineering is often seen as incompatible with agile methods. The primary reason for this is the fact that requirements engineering generates and relies heavily on documentation, while agile methods prefer face to face collaboration with minimal documentation [67].

In essence, agile methods are less document-centric and more code-orientated. Paetch et al. [67] points out that agile methods are adaptive rather than predictive. Traditional approaches plan most of the software effort upfront, in detail, for a large time frame. This works well when requirements are stable and the domain is understood well by the development team. If we however refer back to the findings on project failures (sections 3.2.11 and 3.2.5), many project failures were attributed to factors such as changing business needs and requirements and lack of domain knowledge. Agile methods were developed specifically to adapt and thrive on frequent changes. A dominant idea in agile development is that the team can be more effective in dealing with change if it is able to reduce both the cost of moving information, as well as the time between making a decision and seeing the consequences of that decision [68]. Volatile requirements require a large amount of documentation maintenance. This consumes both time and money - resources agile teams try to conserve. It further becomes clear why the agile community frowns upon a formal requirements engineering process when we remind ourselves of the findings described in section 4.8.6, namely that documentation is often not updated, and outdated documentation is misleading. This could lead to further costs and delays.

Although requirements engineering is not formally included in any of the agile development methodologies, Paetch et al. [67] show that the requirements elicitation, analysis, validation and management processes of requirements engineering is present and practiced on an continuous basis, although they are not as clearly separated. For example, because agile approaches require the customer to be readily available to the development team, the development team is able to elicit requirements directly from the user on a continuous basis. All agile approaches require features or stories to be prioritized, and possible implementations are modeled informally during team design sessions. These practices can be seen as a form of analysis. Requirements validation is performed by the implementation passing acceptance tests as well as feedback provided during frequent review meetings. Finally, the project backlog or feature list provides the base for requirements management as this is what determines the scope for the iteration. It is only when it comes to the documenting of requirements that we note a clear absence.

Although the “lazy” approach to requirements engineering practiced in agile teams have clear cost benefits, Paetch et al. cautions on the long term effects a lack of documentation could have on the organization [67]. In an industry where high staff turnover is often experienced, not having documentation would require existing resources to devote time and effort getting new staff of the ground. Agile teams also rely heavily on “good” developers doing the “right” thing. These developers are often in short supply. It is worth stressing that both staff turnaround and skills shortages were implicated as contributing factors to many failed projects. See Table 3-3.

Eberlein [69] further cautions on the agile approach to requirements engineering. Even though interacting with the customer is seen as a very important contributing factor to project success, he points out that it is almost impossible to get all the necessary requirements from just one person. It is thus still necessary to consider an elicitation process that considers requirements from multiple viewpoints. Another criticism Eberlein has, is around non-functional requirements. He states that it is very dangerous to only consider non-functional requirements at implementation time. Further more, customers find it difficult to articulate their non-functional requirements, so they are often missed if not actively gathered.

4.13 Summary of Requirements Engineering Recommendations

From the literature study it is clear that the process around gathering and managing requirements is of crucial importance. When considering various studies regarding project outcome, there seems to be a direct link between requirements quality and project outcome. The cost of fixing requirement errors are said to compound the later they are found in the development process. Various stakeholders should be consulted in order to obtain a complete set of requirements. Numerous techniques, frameworks and standards exist to aid in the elicitation, documentation, verification and management of requirements. The quality of the obtained requirements can also be measured and validated against a set of quality measures for requirements to ensure that they are adequate.

4.14 Requirements Engineering on Project Evolution

It became evident when interviewing participants and studying project artifacts that Project Evolution faced major challenges related to system requirements. Various participants stated that late, poor and frequently changing requirements necessitated large amounts of rework and caused delays, frustration and missed milestones. While examining project scorecards, risk assessments and progress reports, the delayed business requirements specifications were often found documented as a project risk. "Delayed start due to late delivery of business requirement specifications" were classified as one of the top rated risks from as early as June 2007. It was documented as having a "very high risk" impact on the business with a "highly likely" probability of occurring. As mitigating actions, "monitor daily for compliance, and introduce parallel streams of activity" were documented.

The final business requirements for the Compass administration system were only delivered towards the end of December 2007, well after the original delivery date of the system. The late delivery of requirements ultimately had two major effects on the project. Firstly the original delivery date was changed from end November 2007 to beginning June 2008. Secondly, the project was partitioned into multiple streams that needed to be implemented simultaneously in order to meet the delivery date. It also needs to be said that questions remained over the overall quality of the final signed-off requirements. During the implementation phase, many changes to the requirements were also requested from both the development and business stakeholders.

4.14.1 Requirements Process

At the onset of the project the management team together with the sponsors decided to adopt the Rational Unified Process (RUP) as a system development life cycle methodology. Time and resources were allocated for both the business modeling and requirements workflows. The project schedule also apportioned time for the inception and elaboration phases where most of the activities surrounding system requirements take place. Consultants were contracted to document and map the "as is" business processes. This was done in the Integration Definition for Function Modeling (IDEF) notation. It was the responsibility of the business analysts to meet with various key business users to gather and document the business requirements in Business Requirement Specification (BRS) documents. The BRS documents were then distributed to the appropriate stakeholders and thereafter Joint Application Development (JAD) sessions were scheduled to discuss and analyze the requirements. After the business requirements were signed off by all the stakeholders, the development team would prepare System Requirement Specification (SRS) documents.

Preparing SRS documents allowed the system analysts and senior developers to further assimilate the requirements. They included the architectural decisions and constraints as set out by the architect in the documents. The architect reviewed the SRS documents before they were passed on to the developers. The BRS and SRS documents were finally used by the developers to implement the system.

The project team definitely defined and adopted a requirements engineering process and performed most of the recommended activities. Evidence to this effect could be found both by interviewing participants and examining the project documentation. On further investigation it was however found that the degree to which the project conformed to a formal requirements engineering process degraded over time. The following sections will discuss each recommended activity in turn and the degree to which they were practiced. Thereafter we will discuss some of the shortfalls and challenges experienced by the participants and possible reasons for these.

Requirements Elicitation and Specification

The business analysts were responsible for eliciting and documenting the business requirements. Apart from their existing experience and domain knowledge, they mainly used three sources to gather the business requirements: Firstly, they met with business users to ask them what they require to perform their duties. Secondly, they studied the functionality available in the existing system. Lastly, they consulted the IDEF business process models documented by the consulting firm. The requirements were captured in MS Word documents using the standard company BRS template. The requirements were further augmented with screen designs, prototypes, models and process flow diagrams

Requirements Analysis and Validation

After the requirements were documented and distributed to the various stakeholders, Joint Application Development (JAD) workshops were conducted. The applicable business analyst, development manager, system analyst, senior developers, business representatives and project manager were present on these JAD sessions. During these sessions the group would work through the requirements document and clarify all the requirements. They would attempt to identify any gaps, conflicts and ambiguities and evaluate the feasibility of the requirements. Having identified any of these they would either be dealt with in the workshop, or the project manager would arrange follow-up sessions to address them if no resolution was agreed upon. The business analyst was then responsible for updating the requirements documentation and distributing it for final review before the stakeholders were expected to give a formal sign-off of the requirements.

Requirements Management

It was the responsibility of the project manager to establish a baseline of the requirements and determine which requirements fall within and outside the project scope in order to meet the deadline. He was also responsible for controlling, documenting and communicating any changes made to the project scope. Evidence to this effect was also found in the project documentation in the form of project plans, requirements schedules and change control documents. This was however done mostly at a high level, not specifying specific requirements, but rather parts of the system such as “workflow” or “internal reporting”.

4.14.2 Process Shortfalls

When the participants were asked how closely the delivered system resembled the documented requirements they commented that it was in fact quite different. When the system analyst was asked to offer explanations for these differences she stated that there were many changes made to the requirements and documentation was rarely updated. Proper impact analysis was also never performed. She also reported that developers often misunderstood the requirements and implemented the incorrect things. It was the authors experience that rework was often required at integration points between the different development streams when incompatibilities were discovered.

It was also observed that developers in different development streams often changed requirements amongst themselves in order to render the different parts of the system compatible with each other. Developers also introduced “improvements” to the system at times. These changes were not always communicated back to the systems and business analysts. For this reason the developers were required to “train” the other stakeholders on the use of the system before testing could commence, as they were often the only ones who fully understood the working of it. In some instances the delivered functionality turned out to be contradictory to what business required the system to do. One such example is validation. The developers implemented a number of validation rules on member data, including the fact that ID numbers are a required field. The reasoning behind this was that ID numbers were required to perform entity matching, which was required before the data was uploaded to Compass. The problem was that at the quoting stage the business users often did not have the ID numbers of the members, and thus were unable to load the membership data until this validation was removed.

The systems analyst also believed changing resources to be another source of the multitude of change requests received after sign-off. For the duration of the project, the business area experienced a high degree of staff turnover. The systems analyst said that even though the requirements were clear and agreed upon in the beginning, they kept changing because the people kept changing. She stated that everybody asked for the system to do what they thought it should do rather then what the business needed it to do. One particular example was the manner in which the system calculated the unit rate for benefits. This particular function was reworked a total of four times before it was agreed upon by all the business teams and systems team. The quote document was another such example.

When the project manager was asked why the delivery of the final business requirements were so delayed he attributed it to resourcing problems. The fact was that there were no Compass skills present in either the systems team or the business teams. When resources with these skills were finally secured it was discovered that the requirement documents that the business analyst team was working on up to that point were inadequate and not compatible with “the Compass way of doing things”. Both the existing systems and business team members had to then first be trained to use the Compass platform. Thereafter the business analyst team was moved to the systems area and they completed the requirements together with the newly appointed Compass development team.

The delay in the Compass administration system requirements also caused delays in other development streams, particularly the quoting system. By the time the Compass requirements were signed off, the quoting system team was well into their implementation phase. When it however came to integrating into Compass at the point when a quote is accepted and moved onto the administration platform, many gaps and conflicts were discovered in the original quoting system specifications.

The skills and capabilities of the business analyst team were also often questioned. Shortly after the business analysis team was moved to the systems area, three of the four business analysts resigned with immediate effect. Only one junior business analyst remained. The vacant positions were filled relatively quickly, but the newly appointed BA's had nowhere near the experience and knowledge of their predecessors. This was recognized as a project risk and was so noted in the project artifacts. To compensate for the lack of domain experience in the BA team, an experienced business user was also moved to the systems area and placed in charge of the BA team.

4.14.3 Conclusion

Even though no formal standards were followed during Project Evolution, the project team did attempt to start off on the right foot by defining and following a development methodology which included a requirements engineering process. The process included many of the recommended activities. Nevertheless the overall quality of requirements remained low. Many changes were required well after implementation was underway and even in the months following the delivery of the system. The team also admittedly deviated from the process as the project went on.

At the onset of the study the author expected to find a lack of knowledge of requirements engineering practices and compliance to these practices as possible explanations for the challenges related to requirements. This did not turn out to be the case. A clear process was visible. Even though some of the participants lacked the appropriate skills to conduct proper requirements engineering, others on the team had sufficient knowledge and experience to provide guidance.

It seems that there are two primary reasons for the amount of requirements churn that took place on the project. Firstly, while to project team had sufficient knowledge and experience of requirements engineering practices, they lacked Compass specific domain knowledge. Secondly, as a result of delays caused by the need to rework many of the requirements, time became a scarce resource and the team was forced to cut corners. For this reason many of the activities required to produce and manage high quality requirements were not performed.

In order to make up time, the decision was made to break development efforts up into a further number of parallel development streams. Even within the teams responsible for developing the Compass administration system and Java quoting system development was fragmented. Individual developers were required to analyze and implement individual requirements, which were tightly coupled, simultaneously. The systems analyst reported that many of the developers, working alone, unknowingly misunderstood many of the requirements. When the developers attempted to integrate the separate parts of the system the shortcomings of the requirements were further highlighted when several incompatibilities became evident. At that stage each developer did whatever they needed to do to meet their deadline, which meant that the requirements process was mostly bypassed. The changes made were often not communicated beyond the developers involved. The resulting system was in cohesive and did not function as the users envisioned it to.

Although it is not desirable, requirements are allowed to, and do change over time. This happens because certain information that has an impact on the requirements often only becomes evident at the implementation stage of the project. Any good requirements process should cater for this. It is however when these changes are not managed and communicated that they become a problem.

It can be argued that the Project Evolution team did not spend enough time perfecting the system requirements. The requirements surrounding the system integration in particular could have been improved upon. On the other hand, the activities surrounding eliciting, analyzing and refining requirements have the ability to continue indefinitely depending on the level of detail considered. In the case of Project Evolution however, the requirements phase extended beyond the planned amount of time, not because of the level of detail considered, but rather because requirements had to be reworked when the appropriate Compass skills become available. By spending a disproportionate amount of time on the requirements phase, the other development phases were unrealistically shortened. When placed under time pressure it is human nature to take shortcuts to make up time. On Project Evolution participants took shortcuts by avoiding time consuming meetings and documentation updates. These are unfortunately activities that are central to managing requirements and often they are the only way in which to establish communication amongst project participants.

In conclusion from Project Evolution we can see the necessity of having the appropriate skills available at the onset and throughout the requirements engineering process. We saw that the requirements phases should be conducted efficiently in the planned amount of time. If the requirements phase overruns its schedule, subsequent schedules should be adjusted accordingly. We observed that chaos can erupt quickly if requirements are not managed throughout the duration of the project, especially during the implementation phase.

Although it was not the case on Project Evolution, the implementation phase usually takes up the biggest proportion of the project schedule. Developers are the key players during the implementation phase and usually make up the bulk of the project team. For this reason developers need to be educated on the dangers of requirements churn and gold plating. These should be avoided, and when changes are made they should be dealt with according to the defined requirements process at all times. The process should however be lightweight enough so that it is not perceived as inefficient red tape, while still serving the purpose of communicating and controlling the project scope and status effectively.

Communication is the key objective and benefit of an effective, ongoing requirements engineering process. Project Evolution, and any other software project, only stand to gain from good communication.

5 Architecture Design & Review

Systems often require extensive maintenance and rework due to changes or flaws in their underlying architecture. In our case study, shortcomings in the existing system's architecture, rendering it unable to meet business needs, formed the main motivation for the system rewrite studied. It also became evident that one of the major risks that threatened the project was that the new architecture would not being able to meet performance requirements. It is for these reasons that we will now consider the theory of software architecture and the practices that are recommended to ensure better resulting architectures.

The outline for this chapter is as follows:

Firstly we will attempt to define what is referred to as software engineering, why it is important and what challenges are faced when designing the systems architecture as well as possible strategies to improve the resulting architecture. These topics are addressed in sections 5.1 to 5.4. Quality attributes are often overlooked when building systems as became evident in the case study. Quality attributes are most often addressed by the architecture. Thus it is important to be aware of them while designing the architecture as well as be aware of the design practices, such as architectural styles, patterns and tactics, that aid their implementation. Sections 5.5 and 5.6 address these topics. Section 5.7 will elaborate on the documenting of architectures, as it is important to make the right information available to different stakeholders at different times. This is most often done in practice by making use of architecture views and modeling.

Section 5.8 forms the bulk of this chapter. It discusses the topic of architectural reviews. Architectural evaluations are said to be of great benefit to projects because they identify potential architectural flaws and risks early on in the development life cycle, and thus minimize the amount of rework required at later stages. Section 5.8 will look at a number of evaluation techniques, the costs involved, benefits, best practices and more.

The theory part of this chapter is concluded by a discussion of the two most prominent standards involved in system architecture, namely IEEE Standard 1471 and TOGAF in section 5.9. Finally we will assess Project Evolution from an architectural perspective.

5.1 Defining Software Architecture

As yet there is no one single accepted definition for "software architecture". In IEEE Standard 1471-2000 the IEEE Recommended Practice for Architecture Description of Software-Intensive Systems the word "architecture" is defined as: "The fundamental organization of a system embedded in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."

Another commonly quoted definition is the one from Bass et al. [8]: "The software architecture of a program or computing system is the structure of structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

A shorter, more concise definition by Taylor et al. [80]: "A software system's architecture is the set of principal design decisions made about the system." Taylor et al. elaborates further on the design decisions involved stating that they include design decisions relating to the system's structure, functional behavior, interaction, non-functional properties and implementation.

According to Bass et al. [8], software architecture is the result of technical, business and social influences. These include:

- The system stakeholders. Different stakeholders will have different concerns and goals that they want met by the system, which may be conflicting.
- The developing organization. Staff skills, the development schedule and budget will influence which architecture will be best suited to the system.
- The background and experience of the architect. Architects will tend to lean towards approaches that proved successful in the past when deciding what architecture to implement.
- The technical environment. The technical environment which is current when the architecture is designed will influence the architecture.

Another important term to define is "architectural degradation". Architectural degradation can be described as the difference between a system's prescriptive and descriptive architecture [80]. A system's prescriptive architecture can be seen as the system's as-intended or as-conceived architecture, and need not exist in any tangible form. The descriptive architecture describes how the system has been realized. Architectural degradation comprises of two related phenomena, namely architectural drift and architectural erosion. Architectural drift is defined as the introduction of principal design decisions into the system's descriptive architecture that are (a) not included in, encompassed by, or implied by the prescriptive architecture, but which (b) do not violate any of the prescriptive architecture's design decisions [80]. Architectural erosion is the introduction of architectural decisions into a system's descriptive architecture that violates its prescriptive architecture [80].

5.2 Why is Software Architecture Important

Bass et al. [8] provided a number of reasons why having a defined architecture is important to a software project.

- It is a means for communication and negotiating amongst stake holders and fosters mutual understanding.
- It enables stakeholders to work independently while giving the architect overall responsibility.
- It encourages a review of the requirements early on, providing for a better understanding of the scope of the system.
- It conveys early design decisions.
- It allows the design to be analyzed.
- It conveys constraints on the implementation.
- It dictates component responsibility.
- It provides a basis for a work breakdown structure and resource allocation.
- It helps establish more accurate cost and schedule estimates.
- It enables the system to meet its quality attributes.
- It is a means for assessing risks and consequences associated with implementing changes.
- It makes it possible to develop prototypes and identify problems early in the development cycle.
- It serves as a transferable abstraction of the system which is easier to grasp and reuse.
- It can be used as a basis for training.

5.3 Challenges to Software Architecture

As with software requirements it would seem the number one threat to the architecture of a software system is once again change. Bosch [81] stated that no matter how carefully the architecture is designed, it will have to change at some point. This is typically very costly. Bosch contributes most of the challenges and costs involved in changing the architecture to the vaporization of knowledge. He states that virtually all the knowledge and information concerning the results of domain analysis, architectural styles used, selected design patterns and all other design decisions taken during the architectural design of the system are embedded implicitly in the resulting architecture, but that they lack a first-class representation. Due to this lack of representation the knowledge of the design is quickly lost, and changes to the architecture during the evolution of the system often violates the original constraints. This quickly erodes the design.

Another consequence of a changing architecture stated by Bosch is the fact that obsolete design decisions are often not removed from the implementation. He gives three reasons for this. Firstly because of the effort required to do so. Secondly because of the lack of perceived benefit, and lastly, because of concerns about the consequences. He also states that failing to do so will erode the architecture at a faster rate resulting in even higher maintenance costs and, ultimately, the early retirement of the system.

5.4 Improving the Systems Architecture

Bass et al. [8] lists several process and structural recommendations that may be followed to improve the resulting architecture. These are as follows:

Process recommendations:

- The architecture should be the product of a single person or small group of persons with an identified leader.
- The architecture should have a prioritized list of functional and quality requirements for the system, which the architecture is expected to satisfy.
- The architecture should be documented in an agreed upon notation with at least one static and one dynamic notation.
- The architecture should be circulated & actively reviewed by the stakeholders.
- The architecture should be analyzed and formally evaluated to determine if it will meet the quality requirements.
- The architecture should lend itself to incremental implementation via the creation of a skeletal system in which communication paths are exercised with minimal functionality in order to grow the system incrementally, easing the integration and testing efforts.
- The architecture should result in a specific and small set of resource contention areas, the resolution of which is clearly specified, circulated and maintained.
-

Structural recommendations:

- The architecture should feature well-defined modules whose functional responsibilities are allocated on the principals of information hiding and separation of concerns.
- Each module should have a well-defined interface that encapsulates the changeable aspects of its implementation to allow development teams to work largely independently of each other.
- Quality attributes should be achieved using well-known architectural tactics specific to each attribute.
- The architecture should never be dependant on a certain version of a commercial product or tool.
- Modules that produce data should be separate from modules that consume data.
- For parallel-processing systems, the architecture should feature well-defined processes or tasks that do not necessarily mirror the module decomposition structure.

- Each task or process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
- The architecture should feature a small number of simple interaction patterns. This will increase the understandability, reduce development time, increase reliability and enhance modifiability as well as show conceptual integrity in the architecture.
-

5.5 Quality Attributes

Many systems are redesigned not because they are lacking functionality but rather because they are not scalable, hard to maintain, not secure or not performing adequately. These are the properties of the system referred to as non-functional properties or quality attributes. Taylor et al. defines non-functional properties as a software system constraint on the manner in which the system implements and delivers its functionality [80].

According to Bass et al. [8] no single quality attribute is entirely dependant on the architecture, nor the implementation or the deployment of a system. It is necessary to get the big picture right in the architecture as well as the details in the implementation. They give the following quality attributes that should be considered in the design of the systems architecture:

- Availability
- Modifiability
- Performance
- Security
- Testability
- Usability
- Reliability
- Efficiency
- Scalability

Taylor et al. states that many non-functional properties are qualitative rather than quantitative and often multidimensional. This makes it hard to measure a non-functional property and prove or disprove the extent to which it has been implemented.

While quality attributes remain a tough area to address when building systems, applying good design practices have shown to aid in the achievement of these attributes. A number of these techniques and design tactics are discussed in the following section.

5.6 Designing the Architecture

When designing the systems architecture it is considered wise to build on previous experience and knowledge gained over time. A lot of the best practices in architecture design have been captured in the form of architecture styles, patterns and tactics. Attribute-Driven Design (ADD) is a method that can be used to aid the architect in choosing the correct patterns, styles and tactics for her resulting architecture to fulfill not only its functional but also non-functional or quality requirements.

5.6.1 Architectural Style and Patterns

Software engineers having built different systems across multiple application domains have noticed that, under given circumstances, certain design choices regularly result in solutions with superior properties. Compared to other possible alternatives, these solutions are more elegant, efficient, dependable, evolvable and scalable. Such a set of design choices can be referred to as an architectural style. An architectural style is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system [80]. Examples of architectural styles include object-orientated styles, client-server- , pipe-and-filter-, blackboard-, publish-subscribe-, and even-based styles.

While architectural styles provide general design decisions, they may need to be refined into additional, usually more specific design decisions in order to be applied to a system. Architectural patterns provides sets of specific design decisions that have been identified as effective for organizing certain classes of software systems, or subsystems. An architectural pattern is defined as a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem occurs [80]. Examples of architectural patterns include the State-Logic-Display (Three-tier), Model-View-Controller, and Sense-Compute-Control patterns.

Despite the fact that architectural styles and patterns hold a lot of wisdom and experience, Taylor et al. [80] report that their use is often resisted. As a reason for this they offer the fact that the very nature of adhering to a style or pattern requires the designer to work within a set of constraints. This is often seen as counterproductive or limiting. Taylor et al. [80] however assert that the constraints posed by architectural styles and patterns provide the designer with freedom and effectiveness because:

- Styles restrict one's focus, freeing up thought space and directing attention to the essentials.
- Styles guarantee the applicability of particular analysis techniques.
- The constraints enable the use of code generation and framework implementation.
- Styles provide invariants or assertions the designer can always count on.
- Styles make communication more efficient.
- Styles promote the understanding of design decisions.

5.6.2 Architecture Tactics

Bass et al. [8] writes about a number of tactics that can be applied to architectures to achieve a given quality attribute. He defines a tactic as a design decision that influences the control of a quality attribute response. A collection of tactics is referred to as an architectural strategy. Table 5-1 gives a summary of these tactics, further details on these tactics are beyond the scope of this paper.

Availability Tactics	Fault detection	Ping/Echo
		Heartbeat
		Exceptions
	Fault recovery	Voting
		Active redundancy
		Passive redundancy
		Spare
		Shadow operation
		State resynchronization
		Checkpoint/rollback
	Fault prevention	Removal from service
		Transactions
		Process monitor
Modifiability Tactics	Localize modifications	Maintain semantic coherence
		Anticipate expected changes
		Generalize the module
		Limit possible options
	Prevent ripple effects	Hide information
		Maintain existing interfaces
		Restrict communication paths
		Use an intermediary
	Defer binding time	Runtime registration
		Configuration files
		Polymorphism
		Component replacement
		Adherence to defined protocols
Performance Tactics	Resource demand	Increase computational efficiency
		Reduce computational overhead
		Manage event rate
		Control frequency sampling
		Bound execution times
		Bound queue sizes
	Resource management	Introduce concurrency

		Maintain multiple copies of either data or computations
		Increase available resources
	Resource arbitration	Scheduling policy
Security Tactics	Resisting attacks	Authenticate users
		Authorize users
		Maintain data confidentiality
		Maintain integrity
		Limit exposure
		Limit access
	Detecting attacks	Intrusion detecting systems
	Recovering from attacks	Restoration
		Identification (Audit trails)
Testability Tactics	Input/Output	Record/playback
		Separate interface from implementation
		Specialize access routes / interfaces
	Internal monitoring	Built-in monitors
Usability Tactics	Runtime tactics	Maintain models of the task, user and system
	Design-time tactics	Separate the user interface from the rest of the application

Table 5-1 Quality Attribute Tactics

5.6.3 Attribute-Driven Design (ADD)

Bass et al. [8] describes Attribute-Driven Design (ADD) as a method that can be used for designing the architecture that both satisfy quality and functional requirements. ADD takes a set of quality scenarios as input and employs knowledge about the relation between quality attribute scenarios and architecture in order to design the architecture. ADD is a recursive decomposition process where, at each stage, tactics and architectural patterns are chosen to satisfy a set of quality scenarios and then the functionality is allocated to instantiate the module types provided by the plan. ADD fits into the life cycle after requirements analysis when the architectural drivers are known with some confidence. The output of ADD is the first levels of a module decomposition view of an architecture and other views as appropriate.

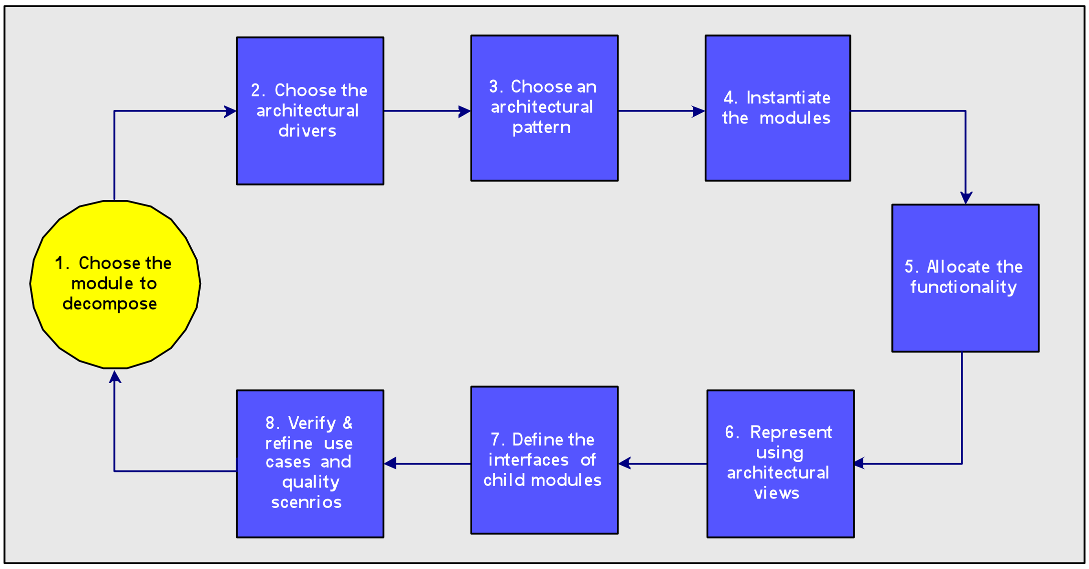

1. Choose the module to decompose
2. Choose the architectural drivers
3. Choose an architectural pattern
4. Instantiate the modules
5. Allocate the functionality
6. Represent using architectural views
7. Define the interfaces of child modules
8. Verify & refine use cases and quality scenrios

Figure 5-1 ADD Process

- **Choose the module to decompose.**
  The system, subsystem and sub-module are all modules. Typically the process starts with the system which is then decomposed into sub systems.
- **Choose the architectural drivers.**
  The architectural drivers can be identified as the combination of functional and quality requirements that shape the architecture. The drivers can usually be found amongst the top priority requirements for the module. Although other requirements apply to the module, choosing the drivers reduces the problem to satisfy the most important ones.
- **Choose the architectural pattern.**
  The goal of this step is to choose an overall architectural pattern that satisfies the architectural drivers by composing selected tactics. The selection of tactics should be guided by the drivers themselves as well as the side effects that the pattern implementing the tactic has on other qualities.
- **Instantiate the modules**
  A system will typically have more then one module: there will be one for each group of functionality. The result of this step is a plausible decomposition of a module.
- **Allocate functionality to the modules.**
  Applying the use cases that pertain to the parent module helps the architect to understand the distribution of functionality and may lead to adding or removing child modules to fulfill al the functionality required. At the end, every use case of the parent module must be represented by a sequence of responsibilities within the child modules.
- **Represent modules and their functionality using architectural views.**
  In most cases 3 views are sufficient to represent the architecture in the beginning although more may be added as needed. These views are the module decomposition view, the concurrency view and the deployment view.

- **Define the interfaces of child modules.**
  For the purpose of ADD the interface of a module need only show the services and properties provided as well as those required. The interaction amongst modules can be discovered by studying the decomposition in terms of structure, dynamism and runtime as described in the architectural views.
- **Verify & refine use cases and quality scenarios as constraints for the child modules.**
  So far various modules have been identified, the decomposition now needs to be verified and child modules need to be prepared for their own decomposition. It needs to be verified that the functional requirements, constraints and quality scenarios imposed on the parent are fulfilled by its children. By the end of this step each child must have its own collection of responsibilities.

ADD as other design methods assume functional requirements and constraints as input but differ in the treatment of quality requirements. ADD requires quality attributes to be expressed as a set of system specific scenarios. Depending on the size of the system various iterations of decomposition is necessary before it will be possible to decide on the specifics of how to implement the system. It will however become apparent early on how the system can be subdivided and work allocated to teams to complete.

5.6.4 Unprecedented Design

Taylor et al. [80] state that it inevitably happens that architects are faced with novel problems at one stage or another to which no pre-defined solutions exist. It might also be the case that even though solutions to a given problem exist, the designer is ignorant of them and has no experience implementing them. To him the problem might then also be novel or new.

The first step the designer should take is to indeed verify whether he is facing a novel problem. If no existing solution strategies are to be found Taylor et al. list a number of strategies that can be employed to discover a solution to the problem. They include:

- Analogy Searching. The idea of analogy searching is to examine other fields and descriptions unrelated to the target problem for approaches and ideas that are analogous to the problem at hand, and formulate a solution strategy based upon that analogy.
- Brainstorming.
- Literature Searching. Literature searching involves examining published material to guide and inspire the designer.
- Morphological Charts. The problem is subdivided into parts, a solution for each subpart is then devised and it is hoped that a solution for the whole is created by combining the sub-solutions.

5.7 Documenting and Modeling the Architecture

Babar and Gorton [83] made the following discovery in their 2009 state of practice study. 83% of respondents documented their architecture using various modeling notations of which UML was the most used. 76% of respondents reported the use of architectural views. The majority (76%) of respondents reported that their organizations include the main architecture requirements as part of the documentation and 67% reported documenting assumptions and constraints. Only 22% of respondents reported the use of design patterns, detailed functional specifications and in-house templates for architecture documentation.

When documenting the architecture it is firstly important to understand who the intended audience is, and what information they are interested in. Secondly the information made available should be grouped accordingly to allow the reader to be able to identify and assimilate the pertinent information without being overwhelmed by irrelevant information.

5.7.1 Stakeholders

Different stakeholders have different needs; they need different kinds and levels of information from the documentation. It is generally not acceptable to have only one architecture document and expect all stakeholders to understand and interpret it in the same way. A suite of documentation should be compiled with a roadmap that will help different stakeholders navigate quickly and efficiently to the information they are interested in. Understanding who the stakeholders are and what their needs are will help guide us in what to document and how to organize it. Table 5-2 lists the different stakeholders and their use of architecture documentation.

Stakeholder	Use
Architect & requirements engineers	To negotiate and make tradeoffs among competing requirements
Architect & designers of constituent parts	To resolve resource contention and establish performance and other kinds of runtime resource consumption budgets
Implementers	To provide inviolable constraints (plus exploitable freedom on downstream development activities)
Testers and integrators	To specify the correct black box behaviour of the pieces that must fit together
Maintainers	To reveal areas that prospective change will affect
Designers of other systems that this one must interoperate	To define a set of operations provided and required, and protocols for their operation

Quality attribute specialists	To provide a model that drives analytical tools such as rate monotonic real-time schedulability analysis, simulations and simulation generators, theorem providers, verifiers etc. These tools require information about resource consumption, scheduling policies, dependencies and so forth. Architecture documentation must contain the information necessary to evaluate a variety of quality attributes. Analysis for each quality attribute has its own information needs.
Managers	To create work teams correspondent to work assignments identified, to plan and allocate project resources and track progress of various teams.
Product line mangers	To determine if a potential new member of the product family is in or out of scope, and if out by how much.
Quality assurance team	To provide a basis for conformance checking, for assurance that implementations have been faithful to the architectural prescriptions.

Table 5-2 Stakeholders and the communication needs served by architecture (Bass et al 2003)

5.7.2 Views and Viewpoints

The concept of views and viewpoints are often used to limit the amount of information made available at a given time to the benefit of the reader. A view is defined as a set of design decisions related to a common concern or concerns. A viewpoint defines the perspective from which the view is taken. [80]

One famous implementation of views is the "4 + 1" View Model by Philippe Kruchten [3]. Kruchten proposes the use of a model composed of 4 views together with scenarios to describe the system's architecture. The views include:

- The logical view, which is the object model of the design (if an object-orientated design method is used). This shows the logical (often software) entities in the system and how they are connected.
- The process view, which captures concurrency and synchronization aspects of the design.
- The physical view, which describes the mappings of the software onto the hardware and reflects its distributed aspect.
- The development view, which describes the static organization of the software in its development environment.

Use cases or scenarios are used to illustrate the use of the architecture and these become the fifth view. Figure 5-2 illustrates the "4 + 1" Model.

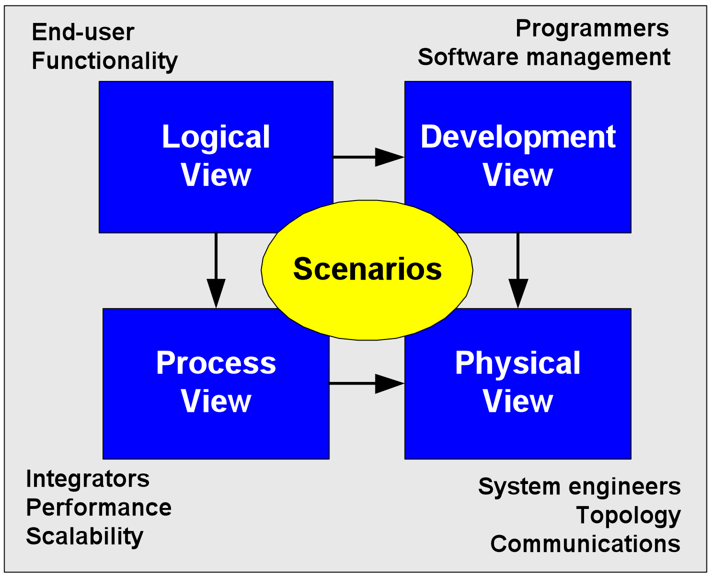


Figure 5-2 The "4 + 1" Model [3]

5.7.3 Modeling

An architectural model is an artifact that captures some or all of the design decisions that comprise a system's architecture. Architectural modeling is the reification and documentation of those design decisions. [80]

Architecture can be modeled in a number of notations, varying from informal, rich, but ambiguous notations such as natural language or box-line diagrams to more formal, semantically narrow architecture description languages such as Rapide or ADML. Taylor et al. [80] categorize and describe a number of notations that can be used to model architecture. These are summarized in Figure 5-3. A detailed discussion of these techniques falls beyond the scope of this dissertation. It is sufficient to say that these techniques vary along many dimensions including what they can model, how precisely they can capture architectural semantics, how good their tool support is, etc. It is up to the architect and project stakeholders to evaluate these methods against their needs and select the notation best suited to their needs.

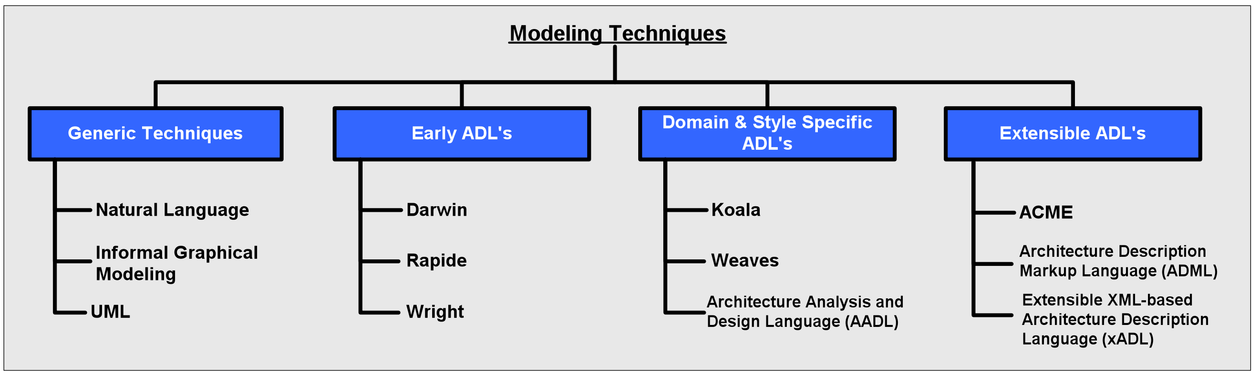

Modeling Techniques
Generic Techniques
Natural Language
Informal Graphical Modeling
UML
Early ADL's
Darwin
Rapide
Wright
Domain & Style Specific ADL's
Koala
Weaves
Architecture Analysis and Design Language (AADL)
Extensible ADL's
ACME
Architecture Description Markup Language (ADML)
Extensible XML-based Architecture Description Language (xADL)

Figure 5-3 Architecture Modeling Techniques [80]

Some systems are simply too complex to model effectively using any of the available techniques. This could be either because they have too many components or because they change too rapidly, making it impossible to know the configuration at any given time. Taylor et al. state that such systems can be modeled to a degree by abstracting to a level where modeling becomes feasible. They suggest the following strategies to consider:

- Model limited aspects of the architecture.
- Model an instance or smaller portion that is relevant.
- Exploit regularity by modeling repeating parts of the system.
- Model the style.
- Model the protocol or "contract" that parts of the system adhere to.

5.8 Analyzing the Architectures

Because most of the costs associated with a system are often incurred during maintenance and addition of new features, analysis of the underlying architecture often adds value in revealing underlying flaws in the design early on, preventing costly changes later in the life cycle.

Taylor et al. [80] defines architecture analysis as the activity of discovering important system properties using the system's architectural models. Throughout the literature the terms architecture analysis, architecture evaluation, architecture review and architecture inspection are often used interchangeably. Generally the system's proposed architecture is considered against the various requirements of the system to establish or measure the architectures fitness.

The Software Architecture Analysis Method (SAAM) was developed in the mid 1990's and was one of the first methods used to evaluate software architectures [10]. SAAM formed the basis for the Architecture Tradeoff Analysis Method (ATAM) developed by the Software Engineering Institute at the Carnegie Mellon University. To date ATAM is the most widely known and practiced formal method of conducting architecture reviews.

While a fair number of papers have been published on the topic of architecture evaluations, it is important to note that this does not seem to be a widely adopted practice and that a lot of the data published seem to be from experience conducting reviews at AT&T or the SEI [84][85][14][82][10][11][91][88].

Bass et al. [82] stated that architecture evaluations can be planned or unplanned. A planned reviews often forms part of the normal development lifecycle and is included in the schedule and budget. It is considered a proactive step in validating the projects direction. Unplanned reviews are usually the result of the project being in serious trouble and taking extreme measures to salvage previous efforts. These kinds of reviews are often perceived negatively by the project members and are seen as a challenge to their technical authority.

Architecture reviews can be performed at different stages of the system's lifecycle. Babar and Gorton [83] reported the following findings regarding the stage at which the organizations they surveyed conducted reviews:

- Early stage (after initial architectural decisions have been made) – 80%
- Middle stage (after elaboration of architecture but before implementation) – 34%
- Post-deployment stage – 15%
- System reengineering stage – 28%
- System acquisition stage – 11%

Kazman et al. [10] states two reasons why it is difficult to evaluate the architecture of a system. Firstly there is no common language used to describe different architectures. Secondly there is no clear way of understanding the architecture with respect to an organization's life cycle concerns or software quality concerns such as maintainability, portability, modularity, reusability and so forth.

5.8.1 Goals & Outputs of Architecture Reviews

Taylor et al. [80] stated 4 goals that the stakeholders may wish to achieve by conducting an architecture analysis. They are: completeness, consistency, compatibility and correctness. We will discuss these in more detail.

When analyzing the architecture for completeness the aim is to verify whether or not the architecture adequately captures all the systems key functional and non functional requirements. This can be described as assessing the architecture's external completeness. The architectures internal completeness also needs to be assessed. This will entail verifying whether all the systems elements have been fully captured with respect to the modeling notation and the system undergoing architectural design.

Consistency is an internal property of the architectural model and is intended to ensure that the different components of the model do not contradict each other. When analyzing the consistency of the model it is necessary to check for name-, interface-, behavioral-, interaction- and refinement inconsistencies.

Taylor et al. [80] describes compatibility as an external property of an architectural model, intended to ensure that the model adheres to the design guidelines and constraints imposed by an architectural style, a reference architecture, or an architectural standard. The ease or difficulty of assessing the compatibility of the architecture is greatly influenced by the level and formality in which the design constraints are captured as it is easier the more formal and complete they are.

Correctness is also an external property of the architectural model. The system's architecture is said to be correct with respect to an external systems specification if the architectural design fully realizes those specifications.

When an architecture is being analyzed there are of course other goals as well that the team might wish to achieve such as early estimation of the systems size, complexity and cost or evaluating the system for opportunities for reusing existing functionality.

Arguably the primary goal of conducting an architecture review is to identify issues that might jeopardize the project. Maranzano et al. [84] listed a number of categories in which issues were found in their experience.

- Problem definition. The problem was not completely or clearly defined.
- Product architecture and design. The proposed solution did not adequately solve the problem.
- Technology. The languages, tools, and components being used to build the system were inadequate.
- Domain knowledge. The team did not have adequate knowledge or experience to solve the problem.
- Process. The process for stating the problem and creating the solution was not systematic, complete, or designed to make development manageable.
- Management controls. The necessary management monitoring capabilities, staffing, controls, and decision-making mechanisms were inadequate.

Avritzer and Weyuker [86] also identified a number of categories in which problems were identified during their study of 50 architecture reviews. The severity of the issues ranged from project affecting to simply observed. The categories, along with the number of issues recorded per severity can be seen in Table 5-3. These results seem to indicate that the four areas where the most issues are typically identified during an architecture evaluation include project management, requirements, design and performance.

Category	**Project Affecting**	**Critical**	**Major**	**Minor**	**Recorded**	**Observed**	**Total**
Project Management	98	95	71	48	54	96	462
Requirements	43	102	103	72	33	72	425
Design	14	29	76	42	59	41	261
Performance	14	56	71	32	34	32	239
Operations administration and maintenance	9	12	95	50	32	29	227
Technology	8	25	41	16	25	30	145
Security	7	20	29	11	14	9	90
Testing	0	1	4	3	2	3	13
Network	0	1	6	1	2	0	10
Other	14	17	18	1	10	16	76
Total	207	358	514	275	265	328	1948

Table 5-3 Problems identified during 50 architecture reviews [86]

The output of an architecture review can also be described in terms of the kinds of risks that it may identify. Kazman and Bass [85] listed 4 broad risk categories in which outputs can emerge from an architecture review including: technical risks, information risks, economic risks and managerial risks. Later on Bass et al. [91] grouped the risks identified during architecture reviews in risk themes. They concluded at the same time that most of the risk themes discovered during an evaluation cover risks that arise from the lack of an activity rather then the incorrect performance of it. Figure 5-4 shows the 15 final risk categories Bass et al. [91] described grouped by risk themes.

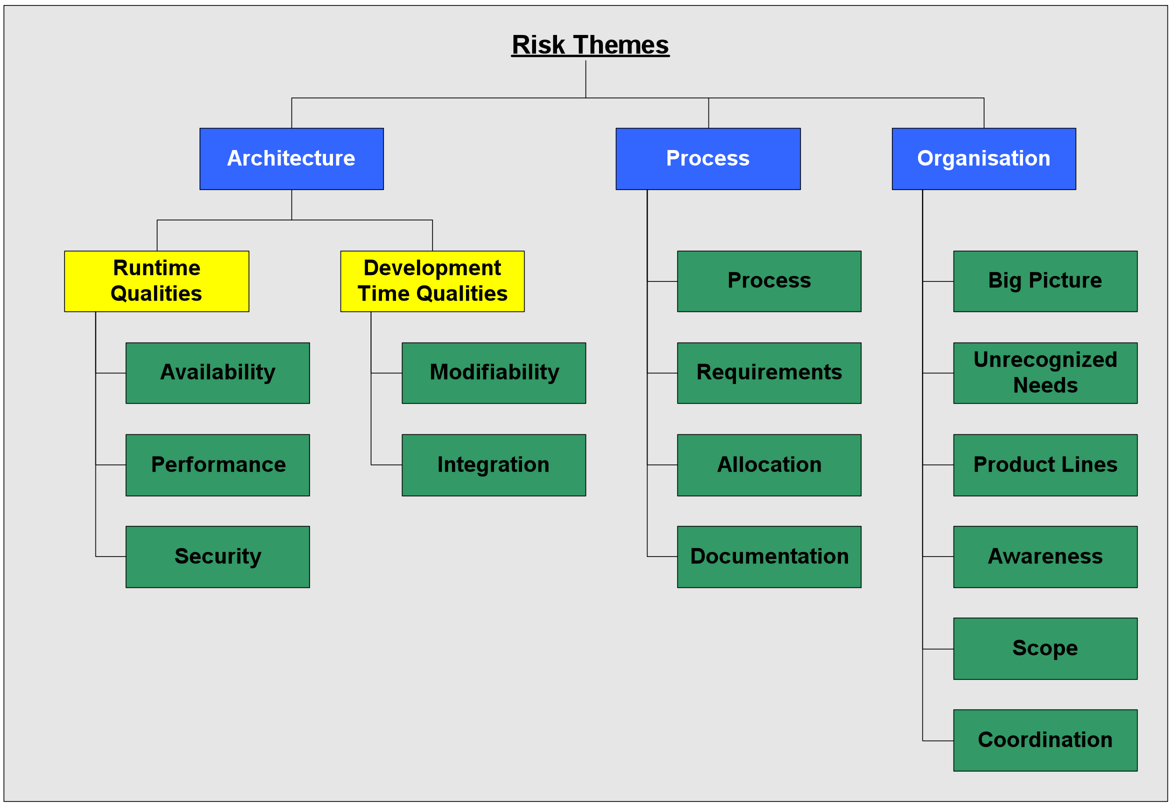


Figure 5-4 Risk Categories [91]

5.8.2 Types of Reviews

Before addressing the different types and techniques available to analyze and evaluate a system's architecture it is important to note that these techniques are directed at different facets of the given architecture. The given technique could for example focus on analyzing either the structural, behavioral, interaction or non-functional characteristics of the system according to Taylor et al. [80].

Architectural analysis techniques can be described as static, dynamic or scenario based techniques.

Taylor et al. [80] describes static analysis as inferring the properties of the system from one or more of its models without actually executing those models. Static analysis can be automated (e.g. compilation) or manual (e.g. inspection). All architectural modeling notations can be analyzed statically whether they are informal box and line diagrams of formal notations.

Dynamic analysis requires execution or simulating execution of the systems model, hence the semantic underpinning of the model must be executable or amendable to simulation. One example of executable models is state-transition diagrams.

The last category of analysis techniques Taylor et al. described [80] is scenario-based analysis. Scenario-based analysis is most appropriate when analyzing large complex systems. Scenario-based techniques typically identify the most important use cases to the system, and then focuses the analysis effort on those.

5.8.3 Stakeholders in Architecture Reviews

Various stakeholders play a role during architecture analysis. Architects, developers, managers, customers and vendors are among the key stakeholders that have an interest in the outcome of the analysis according to Taylor et al. [80].

The review team may consist of members of the project team or it might consist of members external to the project team and include only a few project members. Certain larger organizations have dedicated architecture review teams responsible for analyzing multiple projects within the organization. Obtaining the services of external consultants is another option.

Babar and Gorton [83] discovered in their 2009 study that 29% of the organizations they surveyed made use of dedicated review teams while 68% did not and 3% were unsure. They found that the majority of reviews taking place are performed by the project team themselves and Figure 5-5 illustrates their findings on who typically participated. When asked what factors influenced the selection of participants for architecture reviews 83% of respondents stated that suitable skills influenced the decision. 57% also mentioned experience in architecture reviews, 25% stated organizational position and 12% stated previous performance on reviews influenced the decision.

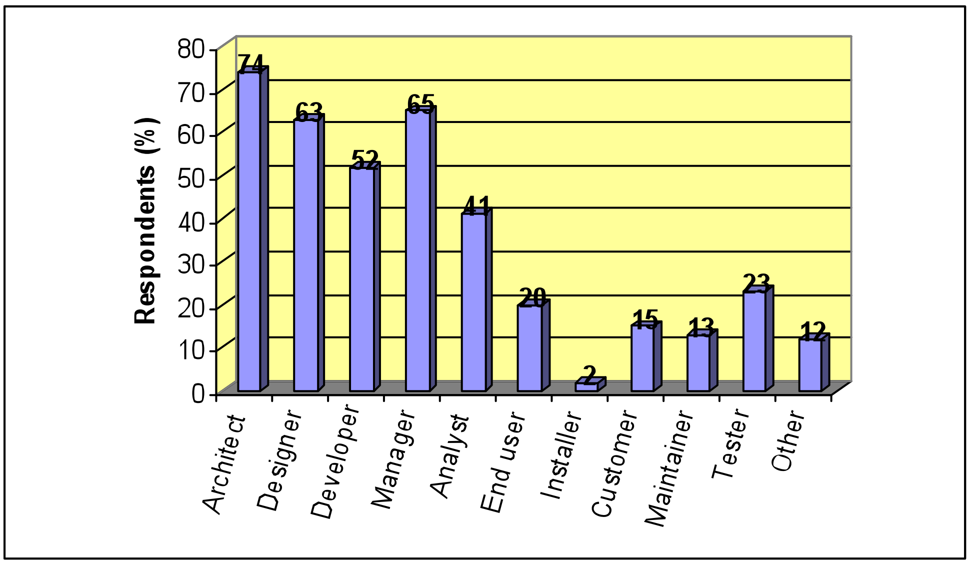


Figure 5-5 Stakeholder participation Babar and Gorton [83]

5.8.4 Architecture Review Techniques

A number of techniques exist to analyze and evaluate system architectures', they can be classified as model-based, simulation-based or inspection-based and review-based techniques according to Taylor et al. [80]. Table 5-4 briefly describes and compares these categories.

	Model-Based	**Simulation-Based**	**Inspection- & Review-Based**
Description	Relies solely on a systems architectural description, and manipulates the description to discover properties of the architecture.	Requires the production of a dynamic executable model of the system or part thereof.	A set of activities conducted to ensure a variety of properties are present in the architecture.
Type of Analysis	Static	Dynamic Scenario-based	Static Scenario-based
Required Model	Formal	Formal	Formal or Informal
Automated/Manual	Partially or fully automated with the use of tools.	Fully automated	Manual
Scope	Individual components & connector. Data exchange Sub systems	Entire system or sub system	Variable
Goals	Consistency Compatibility	Completeness Correctness	Completeness Correctness

	Internally Complete	Consistency Compatibility	Consistency Compatibility
Concerns	Structural properties Behavioral properties Interaction properties	Behavioral properties Interaction properties Non-functional properties	Variable
Stakeholders	Technical Stakeholders (usually the architect)	All	All
Advantages	Less human intensive Less costly Precise	Many different possibilities and scenarios can quickly and cheaply be evaluated.	Can be applied to all models including informal models. Take “soft” properties into account. Take multiple stakeholder objectives and architectural properties into account simultaneously.
Disadvantages	Can only establish “hard” properties or properties that can be encoded in the architecture of the system. Require formal models.	Cannot be applied to all architectural models.	Labor intensive Expensive
Examples	ADL	XTREAM	SAAM ATAM CBAM ARID

Table 5-4 Technique Comparison (according to Taylor et al. classifications)

Bass et al. [82] describes a different classification. They divide evaluation techniques in two categories namely questioning techniques and measuring techniques. Questioning techniques aim to elicit discussion about the architecture and increase understanding of the architecture’s fitness with respect to its requirements. Questioning techniques includes questionnaires, checklists and considering specific scenarios. Measuring techniques are considered more mature then questioning techniques and result in qualitative results. The use of metrics, simulations, prototypes and experiments are all considered measuring techniques. Table 5-5 compares the 2 categories of techniques according to Bass et al. with each other in terms of generality, level of detail, when in the lifecycle the are performed and what is evaluated.

	Review Method	**Generality**	**Level of Detail**	**Lifecycle Phase**	**What is evaluated**

Questioning Techniques	Questionnaire	general	coarse	early	artifact process
	Checklist	domain-specific	varies	middle	artifact process
	Scenarios	system-specific	medium	middle	artifact
Measuring Techniques	Metrics	general or domain specific	fine	middle	artifact
	Prototype, Simulation, Experiment	domain-specific	varies	early	artifact

Table 5-5 Technique Comparison (according to Bass et al. classification)

Babar and Gorton [83] surveyed the use of these evaluation techniques in industry and their findings are shown in Figure 5-6.

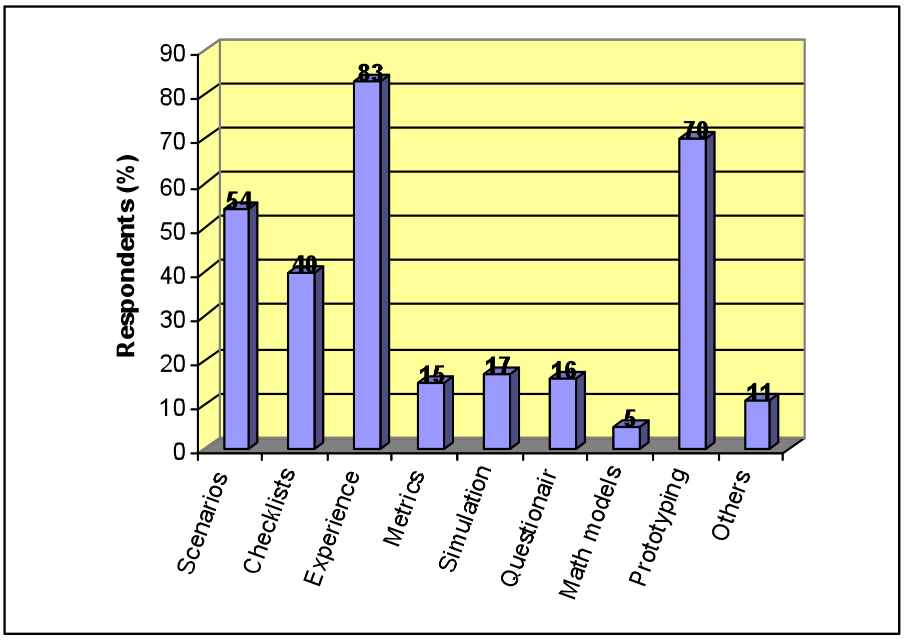


Figure 5-6 Techniques used for architecture evaluation [83]

Babar and Gorton [83] also asked survey respondents what criteria they used to select a particular technique. They found that no respondent mentioned having an organizational policy, standard or guideline for selecting a review technique but identified the following main factors that are considered when choosing a technique:

- Relevant quality attributes for the system under review such as performance, usability and security;
- Size and complexity of the system under review;
- Availability of staff specializing in a certain review technique;
- Development time frame and budget;
- Customer requirements;

- Organizational politics;
- Architect or manager prerogative.

The sections that follow discuss a number of techniques recommended in literature in more detail. Of these the Software Architecture Analysis Method (SAAM) and the Architecture Tradeoff Analysis Method (ATAM) are the most mature and widely known techniques. Interestingly in the Babar and Gorton [83] study only 22% of respondents were aware of these techniques and only 5% of the respondents reported ever using these techniques to perform architecture reviews.

5.8.4.1 Software Architecture Analysis Method (SAAM)

SAAM was the first widely promulgated scenario-based software architecture analysis method [10]. SAAM was created mainly to asses the architectures modifiability. Analyzing an architecture using SAAM starts out with an architecture representation in a notation that everyone involved knows and understands. Secondly you require a set of scenarios depicting all the tasks the system is expected to perform as well as future changes that the analyst foresees. Each scenario then needs to be evaluated against the architecture to determine if the architecture can fulfill the requirement directly of if a change to the architecture is needed to do so. In the case where the architecture is adequate the scenario is referred to as a direct scenario, in the case where changes are required the scenario is referred to as an indirect scenario. It is necessary to note all the changes to the architecture necessary for each indirect scenario along with the costs associated therewith. When two or more components necessitate a change to the same component they are said to interact. The number of interactions will measure the degree to which the architecture supports appropriate separation of concerns as a high number of interactions indicate that the functionality of a particular component is poorly isolated. Each scenario should now be given a weighting relative to it's importance to determine it's overall ranking. With the information gained from this process it should be possible to compare competing architectures. Figure 5-7 illustrates the 5 steps of SAAM and their dependencies.

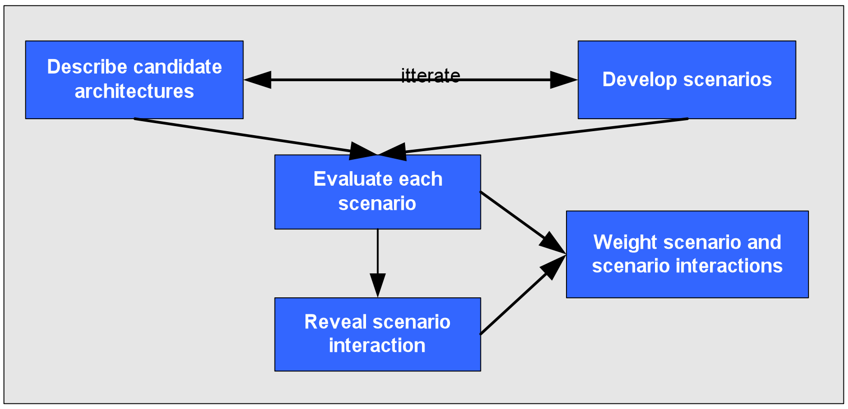

Describe candidate architectures
itterate
Develop scenarios
Evaluate each scenario
Weight scenario and scenario interactions
Reveal scenario interaction

Figure 5-7 Five steps of SAAM and their dependencies.

SAAM participants can be divided into 3 groups, the external stakeholders, internal stakeholders and the SAAM team. The external stakeholders include customers, end-users, marketing specialists, system administrators, maintainers etc. They have no direct involvement in the software architecture development process but are needed to provide the project business goals, the quality attributes along with their expected levels of achievement and the prioritized scenarios. The internal stakeholders have a direct involvement in the architecture and are responsible for analyzing, defining and presenting the architecture as well as estimating the costs and timelines associated with the strategies. Included in the internal stakeholders could be system analysts and the architecture team. The SAAM team in responsible for conducting the evaluation sessions and supporting the stakeholders. The SAAM team consists of a team leader, application domain experts, optional external architecture experts and a secretary.

SAAM's strengths lie in identifying areas of high potential complexity. It is also open for any architectural description but it is not a clear quality metric and is not supported by techniques for performing the various steps.

5.8.4.2 Architecture Tradeoff Analysis Method (ATAM)

ATAM is a scenario-based method for evaluating software architectures. ATAM reveals how well an architecture satisfies particular quality goals and how those goals interact [8][80].

ATAM requires the participation & cooperation of 3 groups. Firstly there is the *evaluation team*, which usually consists of 3 to 5 members internal to the organization or external consultants. They are responsible for conducting the evaluation and making the findings available. The second group that needs to be involved is a set of *project decision makers*. These people should have the authority to speak on behalf of and mandate changes to the project. This may include the project manager and the customer, but should always include the architect. The final group involved represents certain *architecture stakeholders*. These people are directly affected by the architecture promoting various quality attributes. They are the developers, testers, integrators, maintainers, performance engineers, users and others.

ATAM spreads over 4 Phases, during phase 0 the "Partnership and Participation" phase, the evaluation team leadership and key decision makers work out the logistics of the exercise as well as a list of participants and stakeholders. They also agree on what is required by whom and what the outputs will be. This is usually done informally over several weeks prior to the actual evaluation. By the time that Phase 1 and 2 commences the evaluation team would have studies the architectural documentation and have a good idea what they system is about. During phase 1 the evaluation team meets with the project decision makers for about a day to begin information gathering and evaluation. During Phase 2 the stakeholders join the proceedings for further analysis. In phase 3 is follow-up with the evaluation team delivering a final written report and the project team focuses on self examination and improvement. The diagram below summarizes the phases and steps to be performed during the ATAM evaluation.

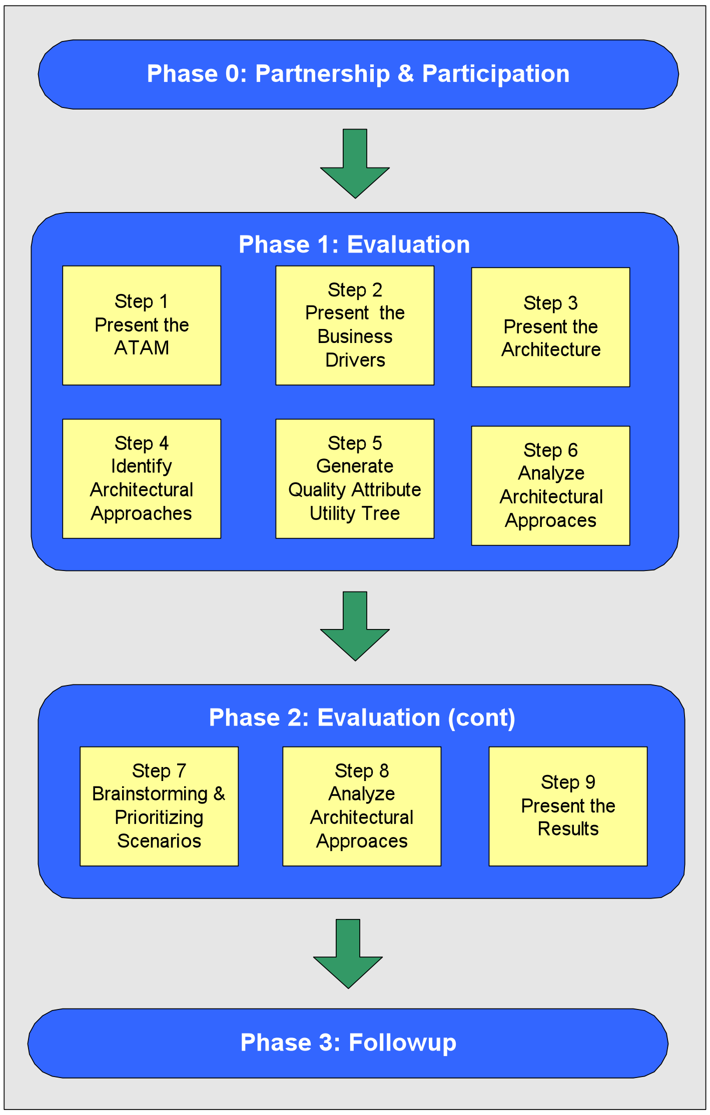

Phase 0: Partnership & Participation
Phase 1: Evaluation
Step 1 Present the ATAM
Step 2 Present the Business Drivers
Step 3 Present the Architecture
Step 4 Identify Architectural Approaches
Step 5 Generate Quality Attribute Utility Tree
Step 6 Analyze Architectural Approaces
Phase 2: Evaluation (cont)
Step 7 Brainstorming & Prioritizing Scenarios
Step 8 Analyze Architectural Approaces
Step 9 Present the Results
Phase 3: Followup

Figure 5-8 ATAM Phases & Steps

Steps	**ATAM Outputs**					
	Prioritized Statement of Quality Attribute Requirements	Catalogue of Architectural Approaches Used	Approach- and Quality Attribute-Specific Analysis Question	Mapping of Architectural Approaches to Quality Attributes	Risk and Non-risks	Sensitivity and Trade-off Points
1. Present ATAM						
2. Present Business Drivers	*a				*b	

3. Present Architecture		**			*c	*d
4. Identify architectural approaches		**	**		*e	*f
5. Generate quality attribute tree	**					
6. Analyze architectural approaches		*g	**	**	**	**
7. Brainstorm and prioritize scenarios	**					
8. Analyze architectural approaches		*	**	**	**	**
9. Present results						

a. The business drivers include the first coarse description of the quality attributes

b. The business drivers presentation might disclose an already identified or long-standing risk that should be captured

c. The architect might disclose a risk in his presentation

d. The architect might disclose a sensitivity or trade-off point in his presentation

e. Many architectural approaches have standard associated risks

f. Many architectural approaches have standard associated standard sensitivities and quality tradeoffs

g. The analysis steps might reveal one or more architectural approaches not identified in step 4, which will then produce new approach-specific questions

Table 5-6 Steps and ATAM Outputs [8]

ATAM has the following strengths; the scenarios are generated based on requirements, it is applicable for static and dynamic properties and produces a quality utility tree. However, it has a weakness in requiring detailed technical knowledge.

5.8.4.3 The Cost Benefit Analysis Method (CBAM)

Economics often play an important role when deciding amongst tradeoffs in a complex system. Given that the resources available to build a system are finite there is a need for a rational process to help decide amongst the different architectural options available. Different options deliver different features and benefits but also require different resources and have varied costs associated with them. It is thus desirable to choose options that have the most benefits relative to the costs that will be incurred to implement it. CBAM is a method that builds on the outputs of ATAM that can be used to determine the optimum option to implement [8].

The ATAM uncovers the architectural decisions made in the system and links them to business goals and quality attribute response measures. The CBAM builds on this base by eliciting the costs and benefits associated with these architectural decisions. The stakeholders can then use this information to decide which option to implement or perhaps invest in implementing another quality attribute all together. The CBAM does not make the decision, but simply provides information the aid in the decision making process. Figure 5-9 illustrates the 9 steps of CBAM [8].

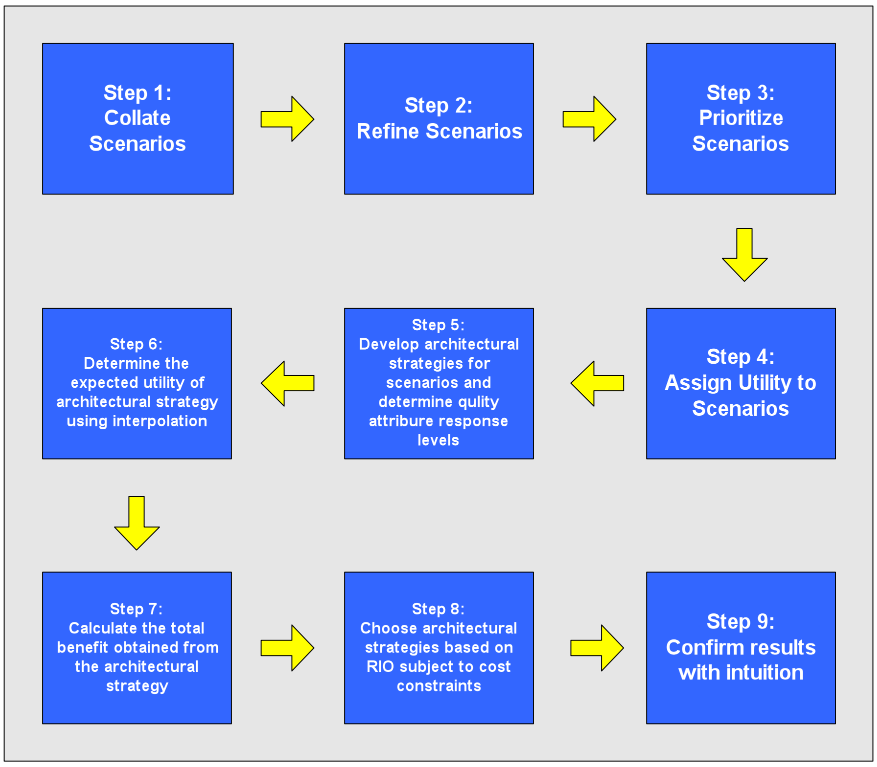

Step 1: Collate Scenarios
Step 2: Refine Scenarios
Step 3: Prioritize Scenarios
Step 4: Assign Utility to Scenarios
Step 5: Develop architectural strategies for scenarios and determine qulity attribure response levels
Step 6: Determine the expected utility of architectural strategy using interpolation
Step 7: Calculate the total benefit obtained from the architectural strategy
Step 8: Choose architectural strategies based on RIO subject to cost constraints
Step 9: Confirm results with intuition

Figure 5-9 CBAM Steps

The basis for CBAM is firstly in considering a collection of scenarios generated by ATAM and assigning a utility to each of them, this is based on the importance of each scenario being considered with respect to its anticipated response value.

Next the various architectural strategies that could be used to implement these scenarios are considered along with possible side affects on other quality attributes. Combining these with the projects cost for implementing the strategy calculates the final return on investment (ROI) measure.

The overall utility of an architectural strategy is calculated across scenarios by summing the utility associated with each one weighted by the importance of the scenario as shown with the formula:

$B_i = \sum_j (b_{i,j} \times W_j)$

Where $b_{i,j}$ is the benefit accrued to strategy I due to its effect on scenario j and W_i is the weight of scenario j. Each $b_{i,j}$ is calculated as the change in utility brought about by the architecture strategy with respect to this scenario: $b_{i,j} = U_{expected} - U_{current}$; that is, the utility of the expected value of the architectural strategy minus the utility of the current system relative to the scenario.

The RIO (Return On Investment) for each strategy is calculated as the ratio of the total benefit Bi to the cost Ci, of implementing it.

$R_i = B_i / C_i$

5.8.4.4 *Architecture-Level Modifiability Analysis (ALMA)*

ALMA was initially developed and tested for Business Information Systems only as a scenario-based evaluation method for software architecture quality attributes focusing on modifiability. Analyzing modifiability usually has one or more of the following goals; predicting costs of changes; identifying system inflexibility and or comparing two or more alternative architectures.

ALMA uses change-scenarios provided by the stakeholders for changes that might occur during the systems evolution. These scenarios are used to verify how well the current architecture can accommodate future changes.

ALMA consists of five steps, not always performed sequentially, and re-iterations over some steps are possible. Firstly an analysis goal is set. This could be risk assessment, maintenance cost prediction or software architecture selection. Secondly a number of architectural views are used to describe the software architecture(s). Then it is necessary to elicit a number of change-scenarios that could affect the architecture in future. These change-scenarios are then evaluated to determine the impact of the change and to express the result in a suitable and measureable way in terms of the chosen goal of the analysis. Lastly the results are interpreted in accordance with the goals of the analysis and verified against the system requirements.

5.8.4.5 *Family-Architecture Assessment Method (FAAM)*

FAAM is used to assess the architecture of information-system families. It focuses primarily on interoperability and extensibility. This is what set it apart from other assessment methods as well as the fact that it actively involves stakeholders from the product-family in the product creation process and emphasizes practical know-how mechanisms and techniques to enable the development teams to implement the method.

FAAM kicks off by defining the goal of the assessment. In order to do so it is necessary to establish the scope and content of the system-family with the stakeholders and the future plans that they have for it. It's also necessary to provide guidelines for the generation and prioritization of requirements. The second step involves the actual generation and prioritization of system quality requirements. Thereafter the architecture needs to be prepared in a representation that can be assessed against the generated requirements. The forth step entails reviewing and refining the artifacts with the goal of producing a set of requirements and artifacts that are relevant and can be used throughout the remaining steps. Next the architecture description is verified against the specified requirements, focusing on the ability to integrate or satisfy the change cases specified in step 2. In the six and final step of FAAM the results are recorded and communicated back to the stakeholders and the facilitator together with the architecture team take in any useful information that arisen from the exercise.

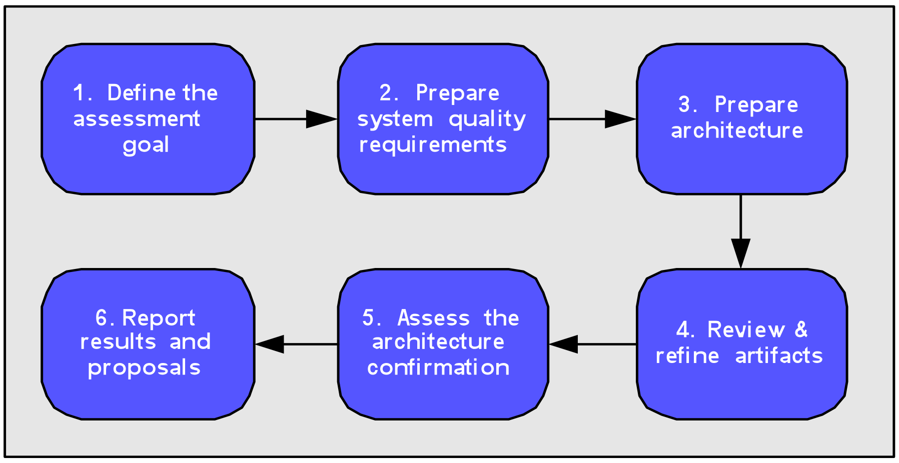

1. Define the assessment goal
2. Prepare system quality requirements
3. Prepare architecture
4. Review & refine artifacts
5. Assess the architecture confirmation
6. Report results and proposals

Figure 5-10 Sequential FAAM steps

Role players in FAAM include the family stakeholders that do not have any direct involvement in the architecture, there role is to present the business context of the project and prioritize the requirements. Business management, product management and development management are examples of family stakeholders. Also included are the internal stakeholders. They are tasked with analyzing, refining and presenting the architecture. Examples of internal stakeholders would be the architect or architecture team. The facilitators have no direct stake in the systems architecture or requirements but rather play a supporting role in the change case generation and architecture presentation. They include a team leader and a number of application domain and external architecture experts as well as administrative and logistical support personal.

5.8.4.6 Active Design Reviews (ADR)

ADR is another methods for ensuring quality, detailed designs in software. This method actively engages reviewers by asking them specific questions regarding a particular design to test their actual understanding of the design [14].

When Pranas and Weiss [87] first described the ADR technique they listed a number of problems with the existing architecture review techniques. These included:

- The reviewers are swamped with information regarding the design, all of which are not necessary to understand the design.
- Most reviewers are not familiar with the goals of the design and the constraints placed on it.
- Responsibility for the review sit with the review team as a whole and not individuals, which in turn results in no part of the design receiving concentrated examination.
- Having a vague idea of their responsibilities or the design goal, the review team may be embarrassed or hesitant to raise concerns over errors or weaknesses.
- Due to the size of the review meetings, interactions between review team members and design team members is limited making detailed discussions hard to pursue.
- Often the wrong people are present at review meetings, turning it into a tutorial.

- Reviewers are often asked to examine issues beyond their competence.
- There is no systematic review procedure and no prepared set of questions to be asked about the design.

In response to these identified problems Pranas and Weiss [87] described ADR as a technique where by a number of smaller review meetings are held, each addressing a specific type of design error, rather then one large review meeting. Because each review meeting targets a specific area of the design, each review requires its own expertise, and therefore only reviewers with specific expertise are involved. The reviews are also not conducted as discussions, but the reviewers are instead provided with a list of questions that they are required to answer using the design documentation. The issues raised by the reviewers as a result of answering the questionnaires are then discussed in small meetings between designers and each reviewer.

ADR's are primarily used to evaluate detailed designs of coherent units of software such as modules or components. The questions asked address the quality and completeness of the documentation and the sufficiency, fitness and suitability of the services provided by the design [14]. The success of the technique lies in the selection of appropriate reviewers and making good use of their skills and time while focusing on specific problems.

5.8.4.7 Active Reviews for Intermediate Design (ARID)

ARID is a hybrid between ADR (Active Design Reviews) and ATAM (Architecture Tradeoff Analysis Method) [14]. It is applied is situations where the stakeholders would like to verify a particular part of the architecture or design before the whole architecture is completed or well documented, as early design evaluation provides valuable insight into the designs viability and early discovery of errors, inconsistencies or inadequacies.

Similar to most design evaluation methods ARID required three groups of participants. Firstly the ARID review team. The second group is represented by the lead designer who is the spokesperson for the design and to whom the results of the ARID will flow. Lastly ARID requires a number of reviewers selected from the stakeholder community. These are people who have a vested interest in the adequacy and usability of the design. They would typically be software engineers who would be expected to use the design. Depending on the size of the user community the aim should be for approximately half a dozen stakeholders.

The ARID process consists of two phases. Phase 1, the pre-meeting phase has four steps and phase 2, the review meeting has a further five steps. These are summarized in the Figure 5-11.

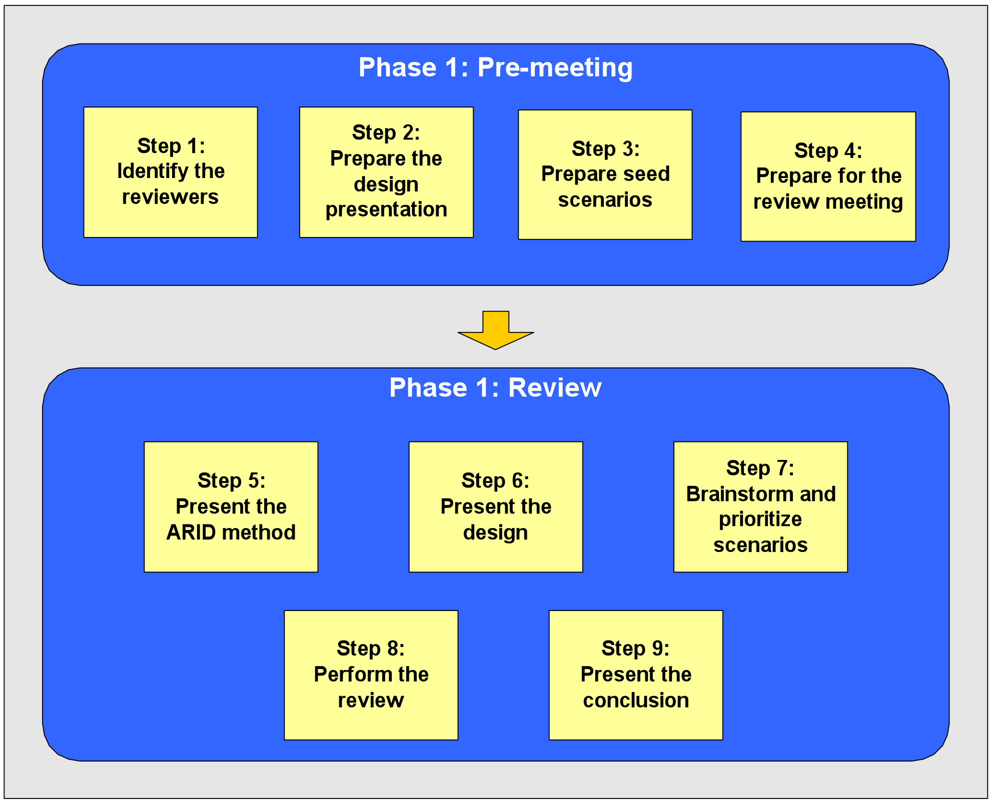

Phase 1: Pre-meeting
Step 1: Identify the reviewers
Step 2: Prepare the design presentation
Step 3: Prepare seed scenarios
Step 4: Prepare for the review meeting
Phase 1: Review
Step 5: Present the ARID method
Step 6: Present the design
Step 7: Brainstorm and prioritize scenarios
Step 8: Perform the review
Step 9: Present the conclusion

Figure 5-11 ARID Phases & Steps

Phase 1, the pre-meeting phase takes the form of a meeting between the review facilitator and the lead designer. The designer needs to prepare a explanation of the design together with a number of examples of it's use. The presentation should have enough details for the reviewers to be able to use the design. The seed scenarios illustrate to the reviewers the concept of a scenario. The seed scenarios may or may not be used in the actual evaluation depending if the reviewers accept or reject them after they brainstormed their own scenarios.

During phase 2 all the reviewers are assembled and the review meeting commences. The review facilitator should spend about 30 minutes explaining the steps of the ARID method. The lead designer should spend approximately 2 hours explaining the design to the reviewers and working through the chosen examples. As in ATAM the reviewers are asked to brainstorm various scenarios for using the design to solve problems. These scenarios together with the seed scenarios are then sorted and merged where applicable and prioritized. The top ranking scenarios are then used to evaluate the design with. To perform the review the participants will be requested to jointly craft pseudo-code to implement solutions for the problems presented in the prioritized scenarios using the examples provided as part of the design presentation. During this step the designer is not allowed to help the reviewers with their solution, he is only allowed to help them navigate to the appropriate part of the design if they have questions on the programs interface or interactions. This step carries on until either the time allocated runs out, all the prioritized scenarios are reviewed or the group feels satisfied that a conclusion on the design was reached. The later being either that the design is usable, indicated by the group quickly understanding how each scenario would be carried out or unusable by the discovery of a show-stopping deficiency.

Because ARID is a hybrid of ATAM and ADR you are able to leverage the strengths of both methods and it is applicable it in situations where neither ATAM nor ADR will prove successful on their own.

Clements provides the following comparison between ATAM, ADR and ARID.

	ATAM	ADR	ARID
Artefact examined	Architecture	Architecture and/or design documentation	Conceptual design approach, with embryonic documentation
Who participates	Architect, stakeholders	Experts in the design realm, consistency and completeness checkers	Lead designer, stakeholders
Basic approach	Elicit drivers and qualities, build utility tree, catalogue approaches, perform approach-based analysis.	Construct review questionnaires, assign reviewer tasks, analyze their results to evaluate quality.	Present design, elicit desired uses, have reviewers as a group use the design.
Outputs	Identified risks, sensitivity points and trade-off points	Errors, gaps and inconsistencies.	Issues and problems preventing successful usage
Approximate duration	3 full days of meetings, over approx. 2 weeks	1-2 days of reviewer work, 1 full day of reviewer de-briefing	1 day pre-meeting and 1 day review meeting.

Figure 5-12 Comparing ATAM, ADR and ARID (Clements 2000)

5.8.4.8 Performance Assessment of Software Architectures (PASA)

Performance failures have been known to result in damaged customer relations, lost productivity for users, lost revenue, cost overruns due to tuning or redesign and missed deadline. According to Williams and Smith [90] most performance failures are due to lack of consideration of performance issues early in the development process, in the architectural design phase. They found that poor performance is more often the result of problems in the architecture rather then the implementation.

Software performance engineering (SPE) is a systematic, quantitative approach to constructing software systems that meet performance objectives [90]. PASA is and extension to SPE methods that address the use of SPE for making architectural tradeoff decisions.

Conceptually PASA resembles the ATAM, but focuses on a single quality of interest namely performance [89]. The performance assessment starts by identifying critical use cases that are important to the responsiveness or scalability of the system. For each of these use cases the scenarios that are important to performance are identified and measureable performance objects are defined for each of those scenarios. At this point the architecture can be analyzed to determine if it is able to support the performance objectives. If there discrepancies are found between the required performance and the ability of the architecture, a decision needs to be made whether to relax the requirement, omit the functionality, increase hardware capabilities or consider alternative architectural designs. The process concludes with a presentation of the results to the clients and an economic analysis of the assessment exercise to justify its cost and highlight the benefits.

PASA can be applied early in the development cycle, post-deployment, or during an upgrade of a legacy system. The figure below illustrates the PASA process model [88].

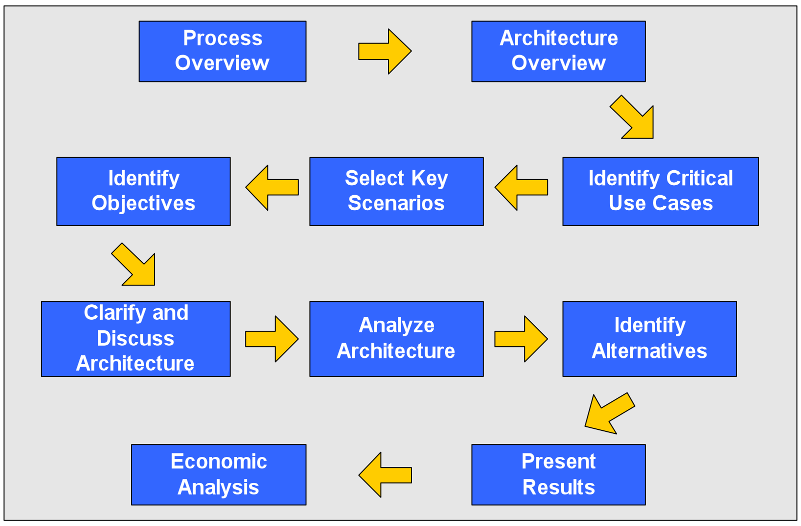


Figure 5-13 PASA Steps

5.8.5 Cost of Performing Architecture Evaluation

The total cost of an architecture review is mainly made up of the costs of the actual review itself as well as the costs incurred in preparing for the review.

When considering the cost of the review itself it is important to factor in the man hours required of both the project team members and the review team taking part. In some cases the review team is only made up of project team members while some organizations might have dedicated review teams working on multiple projects. Depending on what other stakeholders for example end users, take part, it might be necessary to also factor in the cost of their time. The participant’s level of seniority and/or experience will heavily influence the cost of their involvement.

Some organizations make use of external consulting firms to conduct their reviews. In these cases it might be easier to determine the exact cost of the review itself, but it is important to keep in mind that the external review team will not be able to perform the evaluation in isolation and cooperation from resources within the organization will still be required.

The time necessary for the actual architecture review is heavily dependant on the technique used and can be anything from a day to a number of weeks.

Because architecture analysis is dependant on a number of artifacts including the architecture models, system requirements and business goals and objectives a lot of preparation is usually required to not only eliciting information and preparing the documents but also reviewing the documents before the evaluation takes place. The costs related to these should also be considered as part of the overall cost to an architecture review. These artifacts may or may not be routinely required and produced as part of other lifecycle activities such as requirements engineering or design. If they are readily available and of a high enough quality for use in the review the cost of documenting them should not be fully included in the cost of the architecture evaluation.

5.8.6 Benefits to Performing Architecture Evaluations

Bass et al. [82] note the following benefits that can be gained by performing an architecture evaluation:

- Financial Benefits
  Participants in their study reported an average perceived cost saving of 10% over an 8 year period on projects where an architecture evaluation was performed. Architecture reviews proved to control costs while decreasing risk. By preventing unnecessary rework it also has a positive effect on the project schedule.
- Increased Understanding and Documentation of the System
  Performing an architecture evaluation forces the participants to review and document the architecture.
- Detection of Problems with Architecture
  The architecture review process identifies problems associated with unreasonable or costly requirements, performance problems as well as problems associated with downstream modifications etc. so they can be addressed early on in the development process.
- Clarification and Prioritization of Requirements
  A discussion on how well the architecture can meet requirements opens requirements up for discussion. This will result in a clearer understanding of the requirements and their prioritization as well as uncover any potential conflicts and tradeoffs that might need to be negotiated.
- Organizational Learning
  Organizations that performed architecture reviews as a standard part of their development process reported an overall improvement in the quality of their architectures over time. This is because they learned to anticipate the issues that could arise and pre-positioned themselves to maximize their

performance on the reviews. This resulted in not only better architectures after the reviews but also before.

- Improved Overall Quality
  Architecture reviews improved the quality of not only the architecture itself but the overall documentation quality, code quality as well as test quality.

Baber and Gorton [83] also noted the following additional benefits to conducting architectural reviews:

- The review process promotes the capture and documenting for the rational behind important design decisions.
- Promotes confirmation to the organizations quality assurance process.
- Partitions architectural design responsibilities.
- Identifies skills required to implement the proposed architecture.
- Opens new communication channels among stakeholders.

Maranzano et al. [84] stated the following additional reasons why performing architecture evaluations within an organization adds value:

- Leverage experienced people by using their experience to help other projects within the company.
- Let the company better manage software components suppliers.
- Provide management with better visibility into technical and project management issues.
- Rapidly identify knowledge gaps and establish training in areas where errors frequently occur.
- Promote cross-product knowledge and learning.
- Keep experts engaged.
- Spread knowledge of proven practices in the company by using review teams to capture those practices across projects.

5.8.7 Recommendations & Best Practices

Bass et al. [82] made the following recommendations for conducting architecture evaluations:

- Perform planned reviews as part of the normal development lifecycle.
- Perform early, lightweight “discovery reviews” when requirements are known but before architectural decisions are firm in addition to validation review.
- Keep the scope of the evaluation under control.
- Choose an evaluation method that is appropriate to the size of the project to ensure benefits exceed costs.
- During the review process the project should be represented by the architect.
- The evaluation team should ideally be an impartial team external to the development team.
- The evaluation team should contain of members fluent in architecture and architectural issues.
- The evaluation team should contain at least one domain expert.
- The evaluation team should contain someone to take care of logistics.

- The evaluation team should be located close to the source artifacts and have access to domain knowledge, design documents, prototypes, source code, evaluation criteria and support staff.
- Senior managers should set out clear expectations for the review to both the evaluation team and the project personal.
- A clear understanding of the supporting culture within the organization is essential.
- A detailed but flexible agenda is necessary.
- Read-ahead material providing guidance on the evaluation process and questions that will be asked should be provided to the project team by the evaluation team prior to the review.
- The project team must provide read-ahead material to the evaluation team regarding the architecture and rational behind the architectural decisions made.
- The architecture and requirements need to be documented well enough for the evaluation team to gain an understanding of the system.
- Requirements should be ranked to help scope the evaluation and resolve conflicts if they arise.
- All issues that arise during the evaluation need to be recorded and prioritized or classified.
- The evaluation should consider cost consequences of the proposed design to identify possible sources of undue risk and unbound costs.
- The evaluation team should identify clearly where in the architecture the required functionality is performed to ensure the architecture addresses it unambiguously.
- To evaluate performance it is necessary to have work load information, software specification in performance terms and environmental information.
- When serious issues arise they need to be considered both from changing the architecture and changing the requirements point of view.
- All issues raised should be discussed and recorded in a report and feedback provided to the development unit.

5.9 Architecture Standards

Taylor et al. give a number of reasons to adopt standards [80]. Standards can ensure consistency across projects in an organization. Interface or behavioral standards can be used to ensure interoperability or interchangeability between components. Standards are also bearers of engineering knowledge and can carry it from project to project. Standards can reduce the costs of powerful software tools by enabling them to be replicated infinitely at little cost. On the other hand standards also have a major draw back. While stakeholder needs and the desired qualities of a system vary widely from project to project, standards often attempt to apply the same concept and process to all these widely different development efforts. This is then also the reason why in practice only the more general purpose standards tend to be adopted.

In the sections that follow we will discuss 2 architectural standards, namely IEEE 1471 and The Open Group Architecture Framework (TOGAF).

5.9.1 IEEE Standard 1471

IEEE 1471 is and IEEE recommended practice for architectural description. It recognizes the importance of architecture in helping to ensure the successful development of software-intensive systems. IEEE Standard 1471 identifies sound practices to establish a framework and vocabulary for software architecture concepts. Five core concepts and relationships provide the foundation of the standard [92]:

- Every system has an architecture, but an architecture is not a system.
- An architecture and an architecture description are not the same thing.
- Architecture standards, descriptions, and development processes can differ and be developed separately.
- Architecture descriptions are inherently multi-viewed.
- Separating the concept of an objects view from its specification is an effective way to write architecture description standards.

IEEE 1471 places normative requirements on architecture descriptions or documentation. It does not attempt to standardize the actual architecture itself nor the process for developing it. Although the standard primarily focuses on the architecture description, it does not require the use of any specific architecture description language. IEEE takes a stakeholder-driven view of architecture and aims to provide a complete architectural description of the system that will address the concerns of all the stakeholders. It makes use of views and viewpoints to accomplish this. Figure 5-14 illustrates the conceptual model of the IEEE 1471 standard for architectural descriptions.

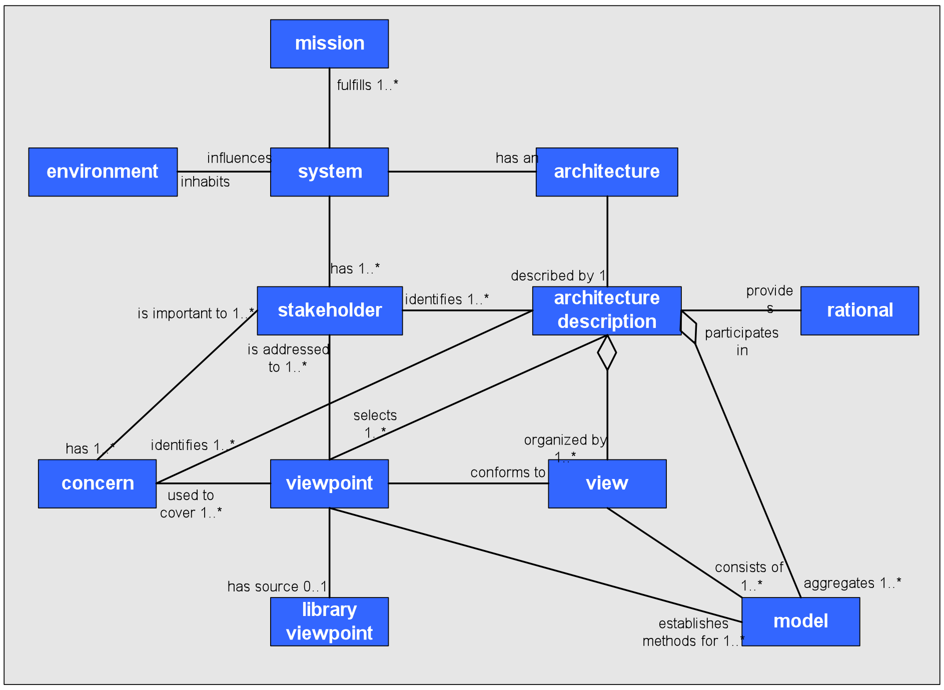


Figure 5-14 The conceptual model of the IEEE 1471 standard for architectural descriptions. [93]

In summary the IEEE 1471 standard requires three things from the architecture description [92]. Firstly the stakeholders and their architectural concerns such as functionality, performance and security need to be identified. Secondly the architecture description needs to be organized into one or more views of the system's architecture. And finally the architecture description must provide the rational for making key architectural decisions.

5.9.2 TOGAF: The Open Group Architecture Framework

TOGAF is the product of a collaborative effort by members of The Open Group in 2003. TOGAF is an "enterprise architecture framework" that takes into account concerns beyond hardware and software, such as human factors and business considerations.

There have been various versions of TOGAF over the years, and the current version, version 8 encompasses business concerns, application concerns, data concerns and technology concerns. TOGAF consists of 3 major elements, as shown in Figure 5-15, the Architecture Development Method (ADM), a "virtual repository" of architecture assets called the Enterprise Continuum, and the resource base [80].

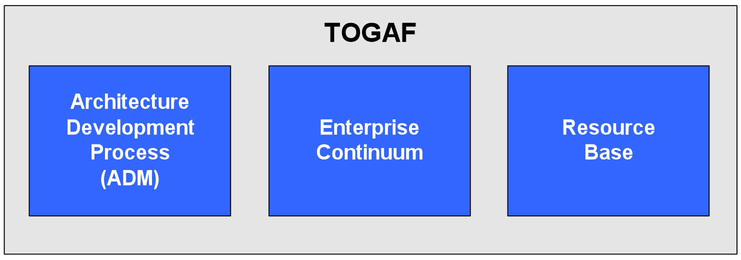


Figure 5-15 Elements of TOGAF

The Architecture Development Process forms the centerpiece of TOGAF. It comprises of a sequence of eight recommended steps and activities (shown below), starting with the architecture vision, that the organization should perform iteratively when developing an enterprise architecture. For each concern different views are prescribed as ways to capture aspects of that concern. The final step, architecture change management, addresses assessment of previous efforts and decides whether it is necessary to perform another iteration of the 8 steps.

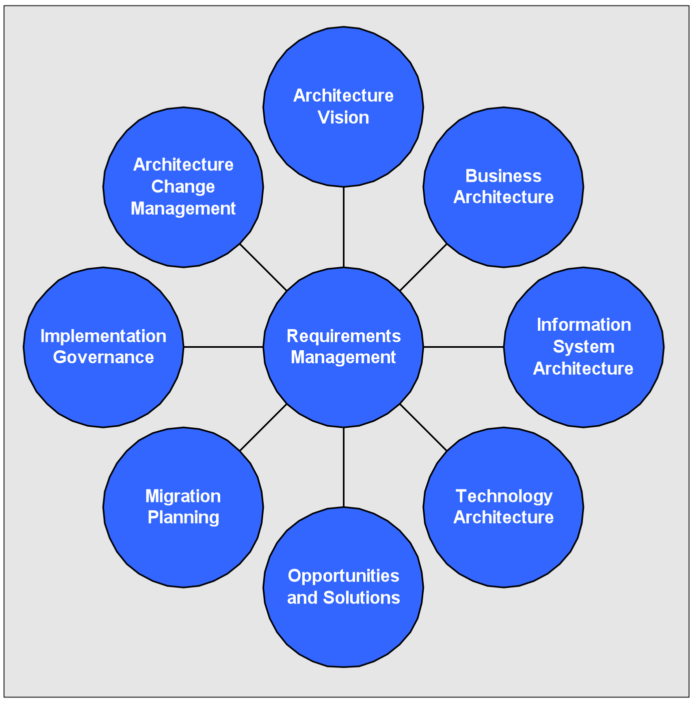

Architecture Vision
Business Architecture
Information System Architecture
Technology Architecture
Opportunities and Solutions
Migration Planning
Implementation Governance
Architecture Change Management
Requirements Management

Figure 5-16 TOGAF Architecture Development Process

The Enterprise Continuum acts as a repository of assets. Included in the Enterprise Continuum is the Architecture Continuum and the Enterprise Continuum as seen below.

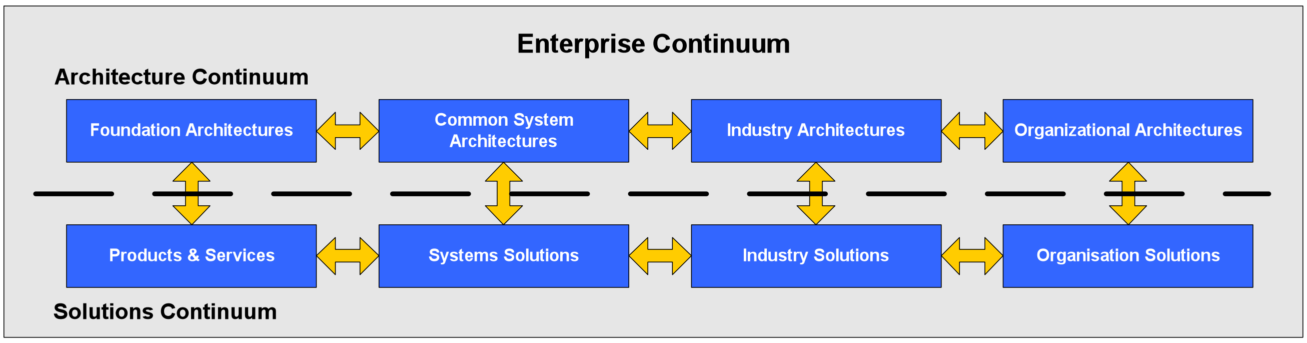


Figure 5-17 TOGAF Enterprise Continuum

The third part of TOGAF is the TOGAF resource base, which is a collection of useful information and resources that can be employed during the ADM process, such as checklists, a catalog of different models and information on how to identify and prioritize different skills needed to develop the architecture etc.

In summary, TOGAF collects a large and diverse set of best practices for architecture. It prescribes not only an architecture development process, but defines a wide variety of views and concerns, as well as solution elements that can be used to satisfy those concerns [80].

5.10 Summary of Recommendations

In summary, we found that the chosen architecture of a system can play an important role in the outcome of a project. When designing an architecture it is important to be aware of the non-functional requirements or quality attributes that the system needs to fulfill and their relative importance. It is often necessary to compromise between these attributes based on the chosen architecture. A number of well documented architectural styles, patterns and tactics exist. These styles, patterns and tactics can help the architect ensure that the resulting architecture meet its non-functional or quality requirements. It is recommended that the architecture be analyzed and reviewed before it is implemented. This aims to ensure that it is able to meet functional and non-functional requirements. This requires the architecture, as well as the functional and non-functional requirements to be well documented. We discovered a number of recommended architecture review techniques that can be made use of, depending on the situation. Lastly, we also identified a number of documented standards and frameworks that attempt to improve the overall process and outcome of software architecture design and implementation.

5.11 Project Evolution and Architecture

It was decided to investigate the practices surrounding the system’s architecture because Project Evolution experienced major performance and scalability problems. System performance and scalability are non-functional requirements that are often heavily influenced by the systems architecture. After the initial version of the system was delivered to the users, poor performance was cited as one of the reasons why the users reverted back to using the old system. The system also encountered problems when trying to produce quotations for organizations with a large number of members.

On the old Globe Quotes system the users were able to calculate quote premiums in a matter of seconds. On the new Evolution Quoting system it could take up to a number of hours to calculate the premiums on a similar quotation. The architectural decision to centralize the calculation engine in Compass was to blame for this. Figure 5-18 & Figure 5-19 below describe the high level architectures of the old and new systems.

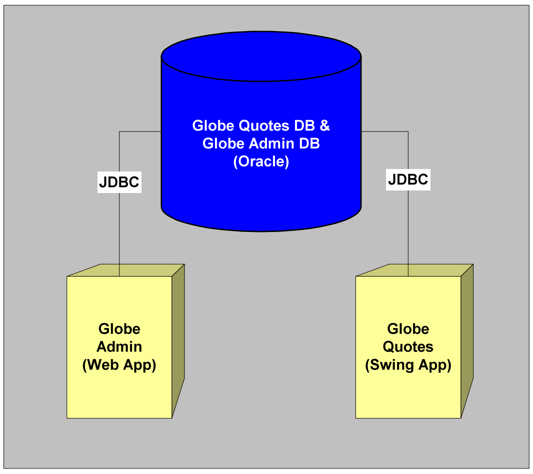


Figure 5-18 High Level Architecture of the Old Globe System

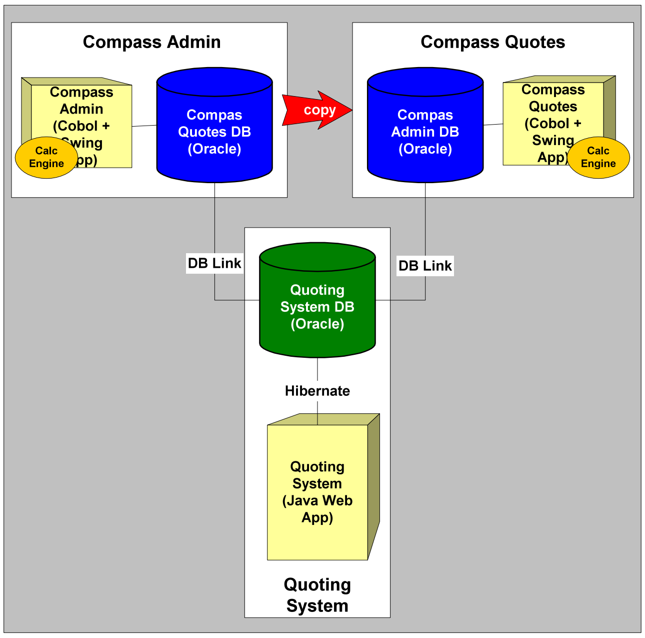

Compass Admin
Compass Admin (Cobol + Swing App)
Calc Engine
Compas Quotes DB (Oracle)
copy
Compass Quotes
Compas Admin DB (Oracle)
Compass Quotes (Cobol + Swing App)
Calc Engine
DB Link
DB Link
Quoting System DB (Oracle)
Hibernate
Quoting System (Java Web App)
Quoting System

Figure 5-19 High Level Architecture of the New Evolution System

The old system's frontends for the quoting and administration systems consisted of a Swing application and web application. Both applications (quoting and administration) shared a single Oracle database that contained both the data and business logic. This made calculating premiums quick, as it simply required a call to a stored procedure.

The new system consisted of separate applications for the quoting and administration systems. The quoting system consisted of a Java web application that also contained the business logic related to capturing quote data. The data was stored in a separate Oracle database. Compass formed the administration system and basically consisted of an Oracle database, Cobol application and Java Swing frontend. The calculation engine to determine the premium that a member needs to pay basically executed a number of sequential steps that needed to be followed for each chosen benefit per member. These steps were configured via the front-end of the application and stored in the database from where they would be invoked by the bill run batch process.

To business, centralizing the calculation engine in Compass sounded like the perfect solution. It promised to speed up development and decrease future maintenance costs, as the premium formulas could easily be configured in one place via the Compass frontend. It did however present a challenge to the systems team. In order to generate premiums in Compass, one had to load the quote as a complete insurance scheme in Compass and then perform a bill run on the scheme. Not only is this a time consuming process, but it meant that each time a quote was changed and recalculated, it would have to generate a new scheme in Compass. This presented a problem in the Compass Administration system, as only one final accepted quote should be loaded as a scheme. To overcome this, a copy of the Compass Administration system was made. This copy was referred to as Compass Quotes. A quote is regarded as still being in process, until the client accepts it. Each time that a quote in progress has to be recalculated, it has to be loaded or "installed" into the Compass Quotes system. Once it was accepted by the client, it is loaded or "installed" into the Compass Admin system. Each time a system change was required to the premium calculations it would be made on the Compass Admin system, and then replicated to the Compass Quotes system, by doing a database copy. In this way the premium formulas were maintained in once place.

Converting a quote to a scheme in Compass in order to perform billing calculations on it was a tedious process. Firstly, all the data required had to be collected from the various tables in the quoting database. Then the data needed to be written in a big staging table on the Compass database, called "interface_entries". Thereafter the "take-on" process had to be started in Compass. This process was scheduled to run every few minutes if unprocessed data were present in the "interface_entries" table and the previous "take-on" process finished running. The "take-on" process would process all the data contained in the "interface_entries" table, copying information (e.g. member and employer information) it to the various tables in the Compass schema. Then it would run through various processes including the billing process to produce premiums for the scheme. Once complete it would place a message on a queue that was monitored by the Quoting system, informing it that the calculation was complete. (Alternatively the message would indicate that a quote had been accepted, and was now installed.) This event would then trigger the Quoting system to copy the resulting premiums from the Compass database and send out an email to the user informing her that the calculation or installation was complete. This process took 2 minutes in the best cases, but depending on the number of members and benefits selected on a quote, the process could take up to 6 hours. When a large quote was being processed, all the subsequent quotes would queue up behind it.

After a few weeks in production, the new business users refused to continue using the new quoting system and reverted to using the old Globe system. This was understandable because generally a single quote needs to be adjusted and recalculated several times before it is send to the client. With a single quote taking between 2 minutes and 6 hours, the system simply could not produce results fast enough if several concurrent users tried to run premium calculations. The users were unable to meet the service level agreements they had with brokers. It was only at this point that the business users indicated what amount of time was acceptable for them to wait for a calculation to complete. It was indicated that calculating a quote should take less then 2 minutes.

Two possibilities were investigated to remedy the problem of inadequate performance. Firstly, the Compass team investigated ways of speeding up the “take-on” process. Secondly, the java team investigated the possibility of implementing a separate calculation engine for the quoting system. It was decided to try and optimize the “take-on” process, but this proved inadequate. The Java team then commenced work on implementing the calculation engine in Java. They were able to get the calculation time down to between 6 and 120 seconds while giving a real time response. The users were happy with the performance and the fact that they no longer needed to wait for an email to carry on with their work. There were however a number of problems, of which the most pressing was the fact that the premiums calculated by the Java calculation engine did not tie up with the ones produced by Compass after the quote was accepted into the administration system. It took a large amount of time and effort to identify and remedy the causes of the discrepancies between the results produced by the two systems. Only after the premiums tiedup did the users start using the new quoting system again.

The system also encountered a problem related to scalability. When the business users attempted to produce a quotation for a 15 000 member scheme, they were unable to do so. The system continuously timed out before completing the member data imports and calculations. The systems team had to manually import the data and perform billing calculations overnight to produce results. Although a number of system changes were made to improve the scalability of the system, schemes of this nature were not very common. For this reason scalability was rated less important than performance and no major system changes were deemed necessary.

5.11.1 Architectural Requirements and Constraints

No formally documented non-functional requirements could be found for Project Evolution. The fact that no non-functional requirements had been defined was raised by the architect as a project risk and documented as such on the project scorecards. Asked why, he replied that: “business was obstructive; they always said that they would do them but never did”. He stated that the non-functional requirements were only addressed from a systems perspective without business input. When asked which non-functional requirements were important to him, the architect replied that maintainability was his most important non-functional requirement. This was followed by extensibility, scalability and testability respectively. When the architect was asked for his opinion on performance he stated that performance is something that can be optimized afterwards, and that it is better to first code for maintainability and then compromise if necessary.

When it came to general requirements that the quoting system architecture had to fulfill, the architect replied that the system had to be web based and use Oracle as a database. It also had to be easily modifiable, and in the future, it had to be accessible by external users via the internet. The chosen architecture also had to facilitate rapid development, since the project originally had to be delivered within a year.

The architect described as an architectural constraint the fact that the quoting system had to make use of Compass as a backend processing system. This decision was made by the management team after receiving input from the Compass development team. He was also instructed to reuse as many components of the existing system and other systems within the company as possible to speed up delivery. He also had to adhere to certain architectural constraints defined by the Discovery Life architecture team.

The architect introduced several further constraints to which the developers needed to adhere. Examples included that the Java code was only allowed to integrate to the backend databases via a OpenJPA persistence layer or stored procedures. For maintainability reasons, JDBC connections and SQL were not allowed within the Java code. All the front end user interfaces were also to be rendered using Java Server Faces (JSF) and all style elements needed to be contained in Cascading Style Sheets (CSS), once again for maintenance reasons. A set of naming and coding standards were also established. Upon later inspection of the code it was found that these constraints were often violated. Junior and new developers joining the team were often guilty of this. Although the team planned to perform code reviews, due to time constraints this did not materialize and the system was delivered with these architectural violations.

5.11.2 Project Evolution Architectural Styles and Patterns

The architect was also asked for his opinion on architectural styles and patterns. He commented that they have positives and negatives. He believed that they are a good tool for explaining and communicating concepts but that there is a risk of over architecting a solution simply to fit a pattern. The architect had a good knowledge of styles and patterns. He also made a concerted effort to share his knowledge and educate the developers on using design patterns by conducting weekly design sessions.

The architect described the systems architecture as a “front-end-back-end” system. Compass provided the backend functionality. The quoting system written in Java was designed using a layered, services oriented architecture. There were 3 main layers defined in the architecture: the domain layer, the services layer and the user interface layer. All the functionality in the system was designed as autonomous loosely coupled services.

5.11.3 Project Evolution Architecture Documentation and Reviews

Only a high-level architecture document could be found in the document repository. It contained detail on the domain, services and user interface layers and what technologies should be used to implement them. It also specified some of the other architectural constraints. A standards and guidelines document containing more implementation specific rules was also found. When asked about architecture documentation, the architect replied that he documented the architecture according to the RUP methodology. It was the author’s experience that most of the design decisions were conveyed to the developers in design meetings and one-on-one conversations with the architect. The developers were asked to document the designs in detail, but these documents were never formally checked into the document repository. They were also not updated when changes were made to the design. In general the architecture was not very well documented.

The architect was questioned on who was responsible for designing the overall architecture, to whom it had to be communicated and how. He replied that he was responsible for the overall design with some input from the Discovery Life architecture team. He had to communicate the architecture to the head of business, the head of systems, the development team and the architecture team. He used the documentation together with presentations and workshops to communicate his design to the stakeholders.

It was established that the architect was familiar with several architecture review techniques, including SAAM, ATAM, ARID, threat modeling and risk driven reviews. In the past he also participated in formal architecture reviews, both as a member of the development team and a member of the review team. He confirmed that no formal architectural review was conducted on Project Evolution. He believed this was mainly due to time constraints. When asked why he believes so few architecture reviews are performed in general, his response was: "They are seen as a waste of time and very often not independent (done by vendors selling a solution). Also most architects have way too much ego to admit they may be wrong. The benefits and need of review are not clearly understood, as software development is seen as a cost centre."

5.11.4 Conclusion

It is possible to debate many of the architectural decisions and technology choices made on Project Evolution. The architect is still of the opinion that choosing Compass as a policy administration system was a mistake. This argument has a degree of merit. When they chose to buy Compass, rather than build an in-house developed system, they believed it would be quicker and cheaper and that the Compass platform was very flexible for future business needs. It turned out that it took much longer than expected to configure Compass and that business had to adapt to Compass rather than Compass to the business. The fact remains that even though it was harder then anticipated, Compass was able to serve as a policy administration platform. The decision to centralize the calculation functionality however proved to be a mistake as a second calculation engine had to be built.

It is impossible to say whether the system would have been more readily accepted by the users if it was able to calculate premiums speedily. There was a noticeable feeling of animosity between the systems and business teams. It is possible that the business users would have identified some other reason for resisting the new system had it performed perfectly. The fact remains that the systems performance was one problem area that could have been avoided if non-functional requirements specification, appropriate architecture documentation, and architectural reviews were included as standard in the development process.

There are a number of recommended practices that were not followed during the architecture design on Project Evolution. Firstly the non-functional requirements of importance to the users should have been addressed formally. These should have included requirements such as performance and scalability. The non-functional requirements should also have been measurable where possible. In the case of performance, the non-functional requirements should have stated the maximum amount of time that a calculation is allowed to take. The project would have also benefitted from formally documenting the architecture in more detail, especially the integration points. If the team had detailed functional requirements and a more detailed documented design available, performing a formal architecture review might have highlighted the performance flaw of the architecture.

Specifying non-functional requirements should be done as part of the requirements engineering process, and it can be argued that their omission can be attributed to the same reasons as the deterioration of the requirements process, namely time pressure, lack of skills and organizational politics. As for the architecture documentation, lack of discipline on the development team's part is partially to blame. The development team members were clearly asked to document their detailed designs and include them in the document repository. Perhaps the fact that they had limited experience in documenting architectural designs, and the architect had limited time to review these documents in detail, also contributed to the poor quality of the designs that were documented. Another reason why design documentation was rarely produced and/or updated is the perception that it is written once and never used again. Perhaps if an architecture review – that would have necessitated documentation – was included as a standard step in the development process, the team would have perceived documentation as more valuable, and made a more concerted effort.

The reasons why an architecture review was not performed unfortunately remains speculative. The most attractive explanation is that the team was overly confident in their solution, and did not perceive that the benefits of performing an architectural review would outweigh the costs of preparing for and conducting one. As stated frequently, the project was also under severe time pressure. This once again affected the architecture negatively. The team could not afford to wait for non-functional requirements to become available, spend time meticulously documenting and updating designs, nor prepare for and conduct a review.

The Compass team was very confident in their decision to centralize the calculation engine within Compass, and convinced management and the architect accordingly. Their confidence was rooted in the fact that centralizing the calculation engine would overcome a challenge that they faced previously while employed at one of Discovery's competitors. At their previous company, they faced many problems due to the fact that the logic used to calculate premiums were duplicated in various systems. This duplication caused these systems to produce different results and required extensive maintenance. Unfortunately while the Compass team had many years of Compass experience, none of them were ever involved in implementing the "take-on" process, and they were unaware of many of the intricacies involved. They also came to Discovery from one of its competitors where asynchronous batch processing is the acceptable means for calculating quote premiums, and the users had very different expectations regarding system performance. So even though it made sense to them to implement a centralized solution, they did not have knowledge of problems related to the alternative, nor knowledge of the performance expectation. They arguably implemented the right solution at the wrong company.

Even though in hindsight an architecture review could have potentially saved more time and money than it would have cost to perform, it would still have required a fair amount of time and money to perform. Seeing as the stakeholders were fairly confident in the chosen architecture, this was time and money they were not willing to spend. The fact that no documented non-functional requirements were available and the architecture was only partially documented contributed to the cost of performing a review. This is due to the fact that these documents are required by most review techniques and would have had to be prepared before hand. It was found that only the architect had prior knowledge of architecture review techniques. The other participants would thus had to have been trained before conducting a review. Formal architecture reviews were also not a common practice within the organization, and no review teams were readily available. While the lack of skills could have been remedied by enlisting the services of an external review team, no schedule or budgetary allowances were made for doing so.

Once again we are able to argue that severely tight deadlines contributed greatly to the shortcomings of the chosen architecture. Firstly the tight schedule enticed the project stakeholders to choose the solution that was the quickest to implement. Secondly, the schedule did not permit the project team to reflect on and review the architectural decisions made. The situation was aggravated by poor communication and working relationships amongst the various participants and stakeholders.

6 Epilogue

While there remain many best practices that were never adopted on Project Evolution, architectural reviews being one of them, several were, such as requirements engineering. However, the degree to which the agreed upon requirements engineering process were followed, degraded over time. I believe that a more diligent implementation of the requirements engineering process as well as conducting architectural reviews would have resulted in a better overall outcome of the project. So why, with all the obvious benefits, did the team not perform an architectural review before the implementation phase commenced? Why did they not continue conforming to the agreed upon requirements engineering process?

Initially, I very much expected to find a lack of knowledge and insight into software engineering theory to be the root cause of the lack of theory adoption in practice. I believed most of the best practices recommended by academics were mostly unknown to the managers and developers working on the project. This turn out not to be the case. The project manager, architect and systems analyst were all fairly well acquainted with many of the methodologies, processes, frameworks and techniques discussed in this dissertation. The architect, in particular, even made a concerted effort to further educate the developers on many of these by means of design sessions. Many of these recommendations were also implemented to some degree.

The next logical reason that I considered, was the possibility that most of the recommended practices were simply not adequate, and not able to deliver positive results. However, while conducting the literature study, I found positive results reported on several projects that implemented these recommended best practices. Perhaps it is more a question of applying the right fix to the right problem. In the case of Project Evolution, the Rational Unified Process (RUP) was chosen as a development methodology for example. RUP is a very popular methodology, and has certainly proved effective on many successful software projects. This being said, perhaps a more agile methodology would have been more effective on Project Evolution. According to the project plan, the requirements and design phase had to be completed during the inception and elaboration phases respectively. This is in line with the RUP methodology. The implicit assumption in this approach, however, is that appropriate domain and technical knowledge is available to successfully complete the task. This was something that Project Evolution struggled to attain. Thus, the outcome of Project Evolution cannot be adduced as evidence that the RUP methodology as a whole is deficient.

Cost is another reason often given for not adopting best practices. Many project teams simply cannot afford to hire expert resources or to buy the fancy expensive tools necessary to follow certain methodologies. This did not seem to be a likely problem on Project Evolution, as several very expensive people were headhunted from a competitor, and large amounts of overtime were authorised. In addition, an external consulting company was hired to map the business process. Furthermore, the company was willing to invest in state of the art hardware and software solutions that promised to improve performance. Taking this into account, together with the huge investment made in the Compass Administration platform itself, the cost of implementing several of the recommended practices would have been negligible. The benefits would have certainly outweighed the costs, and in some instances, practices such as a timely architectural review could even have saved money by preventing unnecessary revisions.

Ultimately I believe that a lack of domain knowledge and technical skills (especially skills related to the Compass platform), together with unrealistic timelines were to blame for both the lack of best practice adoption and for many of the problems experienced on the project. Human nature seemed to further aggravate the situation. This was evident from the amount of "politics" encountered on the project.

The project was initiated shortly after a new general manager (who spearheaded most of the project) was hired in the systems area. Although he had extensive experience in the life insurance industry, he was not intimately familiar with the Compass platform, or the capabilities of the systems and business teams respectively. I believe the decision to implement the administration system in Compass was made without enough information, and that the so called 90% business fit was overestimated. The initial commitment made to the project sponsors of completing the implementation within 12 months was grossly unrealistic and over optimistic.

By the time management realized that they lacked a number of critical skills and took steps to secure resources with those skills, a lot of time had been wasted, and most of the work done up to that point had to be revised. Furthermore, those who had been newly appointed with Compass skills, lacked insight into Discovery's business rules and environment. This caused further delays and misinterpretations.

When the project finally gained traction, the sponsors were unwilling to make further concessions to the schedule. Being placed under pressure to perform, the project participants did what they had to do to protect themselves. These actions, while understandable, were not always with the best possible project outcome in mind. Examples of these actions include: the drastic scope reduction by the systems team; the business team's unwillingness to provide signoff; and their eventual reversion back to the old system. In addition, to save time, changes to requirements were often made by communicating them directly to the developers without proper documentation. This further hampered communication between teams.

Ultimately, for the systems team, making their deadline was all that mattered. Any activity that did not directly add functionality to the system was omitted.

Whether implementing any of the practices advocated by software engineering academics would have been cost effective and would have improved the overall outcome of this project cannot be determined with certainty. But that is not a question that this study set out to answer. Instead, the study merely regarded a number of practices that are said to improve software project outcomes and considered reasons why they were not applied in this particular instance. For the most part it seems that time pressures caused and aggravated by a lack of domain and technical knowledge was to blame.

Hindsight is said to be 20/20. This is also often the case in software engineering. The complex and unique nature of individual software projects, makes it nearly impossible to take all the factors into consideration, and to accurately predict an outcome. Most recommendations are thus formatted within a particular situation, and because of this, their implementation does not always guarantee success under different circumstances. For now, deciding what recommendation to adopt appears to remain part of the art of software engineering rather then science, and art, as beauty remains in the eye of the beholder.

Appendix 1 Research Considerations.

Recommended Principals of Interpretive Research

In their paper, A set of principals for conducting and evaluating interpretive fields studies in information systems, Klein et al. [17] expressed 7 principals that should be followed while conducting, writing up and evaluating interpretive research.

The fundamental principal of the Hermeneutic Circle.
This principal suggests that all human understanding is achieved by iterating between considering the interdependent meaning of parts and the whole that they form.

The principal of contextualization.
Requires that critical reflection of the social and historical background of the research setting, so that the intended audience can see how the current situation under investigation emerged.

The principal of interaction between researchers and subjects.
Requires critical reflection on how the research materials were socially constructed through the interaction between the researchers and participants.

The principal of abstraction and generalization.
Requires relating the idiographic details revealed by the data interpretation through the application of principals one and two to theoretical, general concepts that describe the nature of human understanding and social action.

The principal of dialogical reasoning.
Requires sensitivity to possible contradictions between the theoretical preconceptions guiding the research design and actual findings with the subsequent cycles of revision.

The principal of multiple interpretations.
Requires sensitivity to possible differences in interpretations among the participants as are typically expressed in multiple narratives or stories of the same sequence of events under study.

The principal of suspicion.
Requires sensitivity to possible biases or systematic distortions in the narratives collected from the participants.

Problems associated with case study research.

In their paper, Successfully completing case study research: combining rigor, relevance and pragmatism, Drake et al. [22] described a number of problems associated with case study research. These include:

Designing and scoping of the case study research project in order to ensure the research question is appropriately and adequately answered may be difficult.
Data collection can be time consuming and tedious, and often results in the accumulation of large amounts of data.
Availabilities of suitable case study sites may be restricted.
The reporting of case study research can be difficult as the rigor of the process followed and the validity of the results needs to be established.

Lee [26] also suggests a number of problems with the scientific use of case studies in information system research. They include: (1) Making controlled observations; (2) Making controlled deductions; (3) Allowing for replicability and (4) Allowing for generalizability. He then describes a scientific method grounded in the natural sciences that involves the deductive testing of theories to overcome these problems. In this method four requirements are set out that the theory being tested must satisfy. These requirements are: the theory must be falsifiable, logically consistent, as explanatory or predictive as a competing theory and, lastly, it must survive attempts to falsify it. He states that MIS case studies are capable of achieving scientific objectives through different means such as using natural controls instead of laboratory or statistical controls; using verbal propositions rather then mathematical propositions; replicating a theory's confirmations by applying the same theory to new predictions and, lastly, obtaining generalizability by successive testing across a range of settings.

Guidelines for case study research.

Drake et al. [22] provides several guidelines for successfully conducting case study research.

Considerations in designing, shaping and scoping of a case study project to answer a research question:

- Especially in information systems, the research question should hold value for both practitioners and researchers.
- Designing and scoping a case study research project requires a comprehensive literature analysis to understand the existing body of research literature and position the research questions within that body of knowledge.
- The literature study aids in identifying the appropriate unit of analysis and required number of cases to be studied.
- The unit of analysis must provide sufficient breadth and depth of data to be collected to allow the research question to be adequately answered.

- The number of cases studied depends on the focus of the research question. Single cases provide in-depth investigation and rich description while multiple cases allows for literal or theoretical replication and cross-case comparison.
- The purpose, available resources and deliverables required all impact the scope and design of the case study. For example an MSc dissertation vs a consulting exercise.

To aid the effective and efficient collection of case study data:

Background information should be gathered prior to the commencing of data collection.
Interview time should be used to gather information that cannot be gathered in any other way.
Factual or straight forward information can be collected from other sources, or written answers to structured questions sent to participants prior to the interview sessions.
Well organized and categorized case data will facilitate the task of analyzing the evidence. A case study database should be planned before the data collection commences and material should be documented and organized as it is collected.
Ready access to data should be possible.

Establishing rigor in writing up case study research:

- The researcher must describe in detail how the results were arrived at to establish credibility to the reader.
- To establish validity to the reader the researcher must present a coherent, persuasively argued point of view.
- Sufficient evidence for the research result must be provided.
- Alternative interpretations must be carefully considered and clear reasons must be given for their rejection.
- A general data analysis strategy needs to be developed as part of the case study to indicate what to analyze and why, and to ensure that the data collection activities are appropriate and support the ways the evidence will be analyzed.
- The goal of analysis in interpretive studies in IS is to produce an understanding of the contexts of IS and the interactions between these systems and their contexts.
- Modes of analysis in interpretive studies include Hermeneutics, Narratives and Semiotics.
- The biases introduced by the researcher during the collection and analysis of case data needs to be considered. This included the effects of the researcher's presence at the research site as well as the researcher's values, beliefs and prior assumptions and their influence on the analysis of case study evidence.
- Using multiple sources of data might counteract biases in the researcher's collection and analysis of case data.
- It is important to demonstrate a trail of evidence which the analysis has followed so that the derivation of the case study conclusion is made explicit.

- Because of the volume of data collected writing up case study results can be problematic and the use of recognized case study reporting structures can be useful.
- Case studies should be presented as an interesting and convincing story.
- The case study must be complete and contain sufficient evidence to support the findings.

Appendix 2 Case Study Project Documents

For details, please contact study supervisor.

Appendix 3 Interview Outlines

This appendix contains some of the outlines used to guide the formal interviews with project members.

1. Questions to everyone

1. How would you define a successful project?
2. How would you classify project outcome? E.g. successful, failed, challenged etc.
3. What factors in your opinion typically contribute to the successful outcome of a project?
4. How would you describe a failed project?
5. What factors in your opinion typically contribute to the unsuccessful outcome of a project?
6. How would you describe the outcome or Project Evolution at the following stages? (successful/challenged + why)
7. During the implementation.
8. Directly after delivery.
9. 3 months after delivery.
10. 12 months after delivery.
11. Would you describe Project Evolution as a success or failure overall and what factors contributed mostly to this outcome?
12. Please name and describe 3 major challenges faced during the implementation phases or soon after delivery of PE.
13. Are there any parts of Project Evolution that you would describe as a complete success of failure?

2. Questions to the Project Manager

1. Please tell me about your experience in industry.
2. What are some of the difficulties that you encountered across the different industries?
3. What does project management entail?
4. What was the project vision on Project Evolution?
5. What were some of the key objectives?
6. To what degree were those objectives met?
7. Do you feel everyone understood the objectives?
8. In your opinion was it the right decision to purchase Compass?
9. Tell me about the delivery date.

3. Questions to the Systems Analyst

1. Please describe the requirements gathering process followed during Project Evolution and your role in this process.
2. What requirement elicitation techniques were used?
3. How would you rate the quality of requirements received from business & why.
4. How often did changes need to be made to requirements? Why?
5. Were the requirement documents updated regularly with changes?
6. How closely did the delivered system match documented requirements?
7. If there were any discrepancies between the delivered system and requirements why?

4. Questions to the Architect

1. What were the main architectural requirements in Project Evolution?
2. Were there any non-functional requirements / quality attributes that were of particular importance to you and how did you address them in your architecture design?
3. What architectural constraints did you have to adhere to?
4. Who besides yourself were involved in the architecture design?
5. What is your opinion on architectural patterns, styles and strategies?
6. Did you make use of any architectural strategies / styles / patterns in the architecture design? Please elaborate.
7. Which stakeholders were you required to communicate your architectural decisions to and how did you do so.
8. To what extent did you formally document the architecture?
9. Did you make use of any architectural views?

a. Which ones?

b. Why / why not?

10. What architecture review techniques are you familiar with?
11. Have you ever participated in an architecture review in your career?
12. Did any form of architecture analysis / evaluation / review take place on Project Evolution?

a. If so what were some of the issues / risks / problems identified?

13. In your opinion, why are so few architecture reviews performed in industry?
14. What architecture standards are you familiar with (think IEEE 1471 & TOGAF)?
15. What is your opinion on these standards?
16. With hindsight, what would you have done differently?

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Reference:

[1] Hohmann, Luke. Beyond Software Architecture, Creating and Sustaining winning solutions.

[2] Nagaraj N.S, Srinivas Thonse, Balasubramanian S, Krisnan Narayanan, Kris Gopalakrishnan. Towards a flexible enterprise architecture.

[3] Kruchten, Philippe. Architectural Blueprints – the "4+1" View Model of Software Architecture. IEEE Software 12 (6), November 1995, pp 42-50

[4] Solms, Fritz. The Use Case, Responsibility Driven Analysis and Design Methodology (URDAD),

[5] Leffingwell, Dean. Widrig, Don. Managing Software Requirements, a unified approach. Addison-Wesley 1999.

[6] Westfall, Linda. Software Requirements Engineering: What, Why, Who, When and How. SQP VOL 7, No 4, 2005, ASQ pp 17 – 26.

[7] Davis, Alan, M,. Software Requirements, Analysis and Specification. Prentice-Hall 1990.

[8] Bass, Len, Clements, Paul. Kazman, Rick. Software Architecture in Practice, Second edition. Addison-Wesley, 2003.

[9] Bosch, Jan. Design & Use of Software Architecture, adopting and evolving a product line approach. Addison-Wesley 2000.

[10] Kazman, Rick. Bass, Len. Abowd, Gregory. Webb, Mike. SAAM: A Method for Analyzing the Properties of Software Architectures. 1994.

[11] Kazman, Rick. Bass, Len. Abowd, Gregory. Webb, Mike. Scenario-Based Analysis of Software Architecture. IEEE Software, November 1996.

[12] Babar, Muhammad, Ali. Zhu, Liming. Jeffery, Ross. A Framework for Classifying and Comparing Software Architecture Evaluation Methods. Proceedings of the 2004 Australian Software Engineering Conference. IEEE Computer Society.

[13] Mugurel T. Ionita, Dieter K. Hammer, Henk Obbink. Scenario-Based Software Architecture Evaluation Methods: An Overview.

[14] Clements, Paul C. Active Reviews for Intermediate Designs. Technical Note. CMU/SEI-2000-TN-009. August 2000.

[15] Foote, Brain & Yoder, Joseph. *Big Ball of Mud.* PLoP '97.

[16] Rednor, Hilary (2001). Researching your professional practice, doing interpretive research.

[17] Klein, Heinz and Myers, Michael (1999). A set of principals for conducting and evaluating interpretive fields studies in information systems. MIS Quarterly Vol. 23 No. 1/ March 1999.

[18] Lee, Allen S. (1991). Integrating positivist and interpretive approaches to organizational research. Organizational Science. Vol. 2, No. 4, November 1991.

[19] Garrick, John. Doubting the philosophical assumptions of interpretive research. Qualitative studies in education, 1999, Vol. 12, No. 2 p147 – 156.

[20] Baskerville, Richard. Investigating information systems with action research. Communications of AIS, Vol. 2, Article 19.

[21] Walsham, Geoff. (2005) Doing interpretive research. European Journal of Information Systems.

[22] Drake, Peta. Shanks, Graeme & Broadbent, Marianne. Completing case study research: combining rigor, relevance and pragmatism. Information Systems Journal 8, p273 – p289. 1998.

[23] Benbasat, Izak. Goldstein, David K. Mead, Melissa. The Case Research Strategy in Studies of Information Systems. MIS Quarterly September 1987.

[24] Yin, Robert K. The Case Study Crisis: Some Answers. Administrative Science Quarterly, Vol. 26, March 1981.

[25] Gable, Guy G. Integrating case study and survey research methods: an example in information systems. European Journal of Information Systems 3(2): pp. 112 – 126 1994.

[26] Lee, Allen S. A Scientific Methodology for MIS Case Studies. MIS Quarterly, Vol. 13, No. 1, pp. 33-50, Mar 1989

[27] Robbins, Stephen P. Organizational Behavior. 9th Edition, Prentice Hall. 2001.

[28] Reel, John S. Critical Success Factors in Software Projects. IEEE Software, May/June 1999.

[29] Hoch, Detlev J. Secrets of Software Success.

[30] Agarwal,Nitin. Rathod, Urvashi. Defining 'success' for software projects: An exploratory revelation. International Journal of Project Management 24, p358 – 370, 2006.

[31] Shenhar, Aaron J. Levy, Ofer. Mapping the dimensions of project success. Project Management Journal, Vol 28, June 1997.

[32] Wateridge, John. How can IS/IT projects be measured for success? International Journal of Project Management. Vol. 16, p59 – 63. 1998.

[33] Procaccino, Drew J. Verner, June M. Lorenzet, Steven J. Defining and contributing to software development success. Communications of the ACM, Aug 2006/Vol. 49, No. 8.

[34] Detlev J. Hoch, Cyriac R. Roeding, Gert Purket, Sandro K. Lindner. Secrets of Software Success. Harvard Business School Press. 1999.

[35] Glass, Robert L. Software Runaways. Prentice Hall.1998.

[36] Report on the NATO Software Engineering Conference 1968, Garmisch, Germany.

[37] Glass, Robert L. The Standish Report: Does it really describe a software Crisis. Communications of the ACM. August 2006/Vol. 49, No. 8

[38] The Standish Group. The Standis Group Report, Chaos. 1995

[39] Guide to the Software Engineering Body of Knowladge. 2004 Version. IEEE Computer Society.

[40] Dijkstra, Edsger W. The Humble Programmer. Communications of the ACM, Oct 1972, Vol. 15 No.10

[41] Brooks, Frederic P. No Silver Bullet. Information Processing. Proceedings of the 10th World Computing Conference. 1986

[42] Brooks, Frederic P. The Mythical Man Month. Anniversary Edition. Addison Wesley. 1995.

[43] Molokken, Kjetil. Jorgensen, Magne. A Review of Surveys on Software Effort Estimation. Proceedings of the 2003 International Symposium on Empirical Software Engineering.

[44] Yeo, K. T. Critical failure factors in information system projects. International Journal of Project Management Vol. 20, p241 – p246. 2002.

[45] Glass, Robert L. Evolving a new theory of project success. Communications of the ACM. Nov 1999, Vol. 42 No. 11.

[46] Xia, Weidong. Lee, Gwanhoo. Grasping the complexity of IS development projects. Communications of the ACM. May 2004. Vol. 47, No. 5.

[47] Nolen, Andrew J. Learning from success. IEEE Software. January/February 1999.

[48] Pinto, Jeffrey K. Mantel, Samuel J. Jr. The causes of project failure. IEEE Transactions on Engineering Management. November 1990. Vol. 37, No. 4.

[49] Hall, Tracy. Beecham, Sarah. Verner, June. Wilson, David. The impact of staff turnover on software projects: The importance of understanding what makes software practitioners tick. SIGMIS-CPR. April 2008.

[50] Berntsson-Svensson, Richard. Aurum, Aybuke. Successful software project and products: An empirical Investigation. ISESE Septmber 2006.

[51] Boehm, Barry. Project termination doesn't equal project failure. Computer. September 2000.

[52] Enders, Albert. Rombach, Dieter. A Handbook of Software and Systems Engineering, Emperical Observations, Laws and Theories. Addison Wesley. 2003.

[53] Lewis, James P. Fundamentals of Project Management. Second Edition. Amacom 2002.

[54] Charette, Robert N. Why Software Fails. IEEE Spectrum

[55] Glass, Robert. L. Software Runaways – Some surprising findings. The Journal of Systems and Software 41. 1998

[56] Addison, Tom. Vallabh, Seema. Controlling Software Project Risks – an Empirical Study of Methods used by Experienced Project Managers. Proceedings of SAICSIT 2002.

[57] Karlsson, Lena. Dahlstedt, Asa. Regnell, Bjorn. Natt och Dag, Johan. Persson, Anne. Requirements engineering challenges in market-driven software development – An interview study with practitioners. Information and Software Technology. Vol 49. 2007.

[58] Lubars, Mitch. Potts, Colin, Richter, Charles. A review of the state of the practice in requirements modeling. Proceedings of the 8th International Workshop on Requirements Engineering, Germany. 2002.

[59] Curtis, Bill. Krasner, Herb. Iscoe, Neil. A field study of the software design process for large systems. Communications of the ACM. Vol. 31, No. 11, November 1988.

[60] Emam, Khaled. Madhavji, Nazim. A field study of requirements engineering practices in information systems development. Proceedings of the 2nd IEEE International Symposium on Requirements Engineering, York, UK. 1995.

[61] Kamsties, Erik. Hormann, Klaus. Schlish, Maud. Requirements engineering in small and medium enterprises: State-of-the-practice, problems, solutions and technology transfer. Conference on European Industrial Requirements Engineering. London, UK. 1998.

[62] Hofmann, Hubert, F. Lehner, Franz. Requirements engineering as a success factor in software projects. IEEE Software. July/August 2001.

[63] Decker, Bjorn. Ras, Eric. Rech, Jorg. Jaubert, Pascal. Reith, Marco. Wiki-Based Stakeholder Participation in Requirements Engineering. IEEE Software. March/April 2007.

[64] Lethbridge, Timothy. Singer, Janice. Forard, Andrew. How Software Engineers Use Documentation: The State of Practice. IEEE Software. November/December 2003.

[65] Gotel, Olly. Morris, Stephen. More then just "Lost in Translation". IEEE Software. March/April 2009.

[66] Singh, Yogesh. Sabharwal, Sangeeta. A Systematic Approach to Measure the Problem Complexity of Software Requirement Specifications on an Information System. Information and Management Sciences. Vol. 15, No. 1 2004.

[67] Paetsh, Frauke. Eberlein, Armin. Maurer, Frank. Requirements Engineering and Agile Software Development. Proceedings of the 12th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises. 2003.

[68] Cockburn, Alistair. Highsmith, Jim. Agile Software Development: The People Factor. Software Management. November 2001.

[69] Eberlein, Armin. Sampaio do Prado Leite, Julio, Cesar. Agile Requirements Definition: A View from Requirements Engineering.

[70] Davis, Alan M. Hickey, Ann M. Requirements Researchers: Do We Practice What We Preach? Requirements Engineering. Vol. 7. 2002.

[71] Linscomb, Dennis. Requirements Engineering Maturity in the CMMI. The Journal of Defense Software Engineering. December 2003.

[72] CMMI for Systems Engineering and Software Engineering, Version 1.1. CMU/SEI-2002-TR-001. December 2001.

[73] CMMI for Development, Version 1.2. CMU/SEI-2006-TR-008. August 2006.

[74] Performance Results from Process Improvement. SoftwareTech. Vol. 10 No. 1 March 2007. Software Engineering Institute, Carnegie Mellon.

[75] Emam, Khaled El. Birk, Andreas. Validating the ISO/IEC 15504 Measure of Software Requirements Analysis Process Capability. ERB-1072 National Research Council Canada. February 1999.

[76] Pollice, Garry. Using the Rational Unified Process for Small Projects: Expanding Upon eXtreme Programming. Rational Software White Paper. TP183,301.

[77] Sawyer, Pete. Sommerville, Ian. Viller, Stephan. Capturing the Benefits of Requirements Engineering. IEEE Software. March/April 1999.

[78] Damian, Daniela. Chisan, James. Vaidyanathasama, Lakshimanarayanan. Pal, Yogerenda. An Industrial Case Study of the Impact of Requirements Engineering on Downstream Development. Proceedings of the 2003 International Symposium on Empericl Software Engineering. IEEE Computer Society 2003.

[79] Bosch, Jan. Software Architecture: The Next Step. EWSA 2004. pp. 194-199, 2004.

[80] Taylor, Richard, N. Medvidovic, Nenad. Dashofy, Eric, M. Software Architecture, Foundations, Theory, Practice. Wiley. 2009.

[81] Bosch, Jan. Software Architecture: The Next Step. EWSA 2004. pp194 – 199. 2004.

[82] Bass, Len. Clements, Paul. Kazman, Rick. Northrop, Linda. Zaremski, Amy. Recommended Best Industrial Practice for Software Architecture Evaluation. CMU/SEI-96-TR-025. January 1997.

[83] Babar, Muhammad, Ali. Gorton, Ian. Software Architecture Review: The State of Practice. Computer. July 2009.

[84] Maranzano, Joseph, F. Rozsypal, Sandra, A. Zimmerman, Gus, H. Warnken, Guy, W. Wirth, Patricia, E. Weiss, David, M. Architecture Reviews: Practice and Experience. IEEE Software. March/April 2005.

[85] Kazman, Rick. Bass, Len. Making Architecture Reviews Work in the Real World. IEEE Software January/February 2002.

[86] Avritzer, Alberto. Weyuker, Elaine, J. Metrics to Assess the Likelihood of Project Success Based on Architecture Reviews. Emperical Software Engineering, 4, 199-215. 1999.

[87] Pranas, David, L. Weiss, David, M. Active Design Reviews: Principals and Practices. Proceedings, Eight International Conference on Software Engineering. 132-136. 1985.

[88] Babar, Muhammad, Ali. Gorton, Ian. Comparison of Scenario-Based Software Architecture Evaluation Methods. Proceedings of the 11th Asia-Pacific Software Engineering Conference. 2004.

[89] Bahsoon, Rami. Wolfgang, Emmerich. Evaluating Software Architectures: Development, Stability, and Evalution.

[90] Williams, Lloyd, G. Smith, Connie U. Performance Evaluation of Software Architecture. ACM. 1998.

[91] Bass, Lenn. Nord, Robert. Wood, William. Zubrow, David. Risk Themes Discovered Through Architecture Evaluations. Technical Report. CMU/SEI-2006-TR-012. September 2006.

[92] Maier, Mark, W. Emery, David. Hilliard, Rich. Software Architecture: Introducing IEEE Standard 1471. Computer. April 2001.

[93] Koning, Henk. Van Vliet, Hans. A Method for Defining IEEE Std 1471 viewpoints. The Journal of Systems and Software. No 79. 2006.