Architecture-Centric Project Estimation

Troy S. Henry

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
Computer Science & Applications

Dr. Shawn Bohner, Chairman
Dr. James (Sean) Arthur
Dr. Deborah Tatar

Keywords:
software engineering, estimation, risk analysis, software architecture

May 14, 2007
Blacksburg, Virginia

Copyright 2007, Troy S. Henry

Architecture-Centric Project Estimation

by

Troy S. Henry

(ABSTRACT)

In recent years studies have been conducted which suggest that taking an architecture first approach to managing large software projects can reduce a significant amount of the uncertainty present in project estimates. As the project progresses, more concrete information is known about the planned system and less risk is present. However, the rate at which risk is alleviated varies across the life-cycle. Research suggests that there exists a significant drop off in risk when the architecture is developed. Software risk assessment techniques have been developed which attempt to quantify the amount of risk that varying uncertainties convey to a software project. These techniques can be applied to architecture specific issues to show that in many cases, conducting an architecture centric approach to development will remove more risk than the cost of developing the architecture. By committing to developing the architecture prior to the formal estimation process, specific risks can be more tightly bounded, or even removed from the project.

The premise presented here is that through the process of architecture-centric management, it is possible to remove substantial risk from the project. This decrease in risk exceeds that at other phases of the life-cycle, especially in comparison of the effort involved. Notably, at architecture, a sufficient amount knowledge is gained by which effort estimations may be tightly bounded, yet the project is early enough in the life-cycle for proper planning and scheduling. Thus, risk is mitigated through the increase in knowledge and the ability to maintain options at an early point. Further, architecture development and evaluation has been shown to incorporate quality factors normally insufficiently considered in the system design.

The approach taken here is to consider specific knowledge gained through the architecting process and how this is reflected in parametric effort estimation models. This added knowledge is directly reflected in risk reduction. Drawing on experience of architecture researchers as well as project managers employing this approach, this thesis considers what benefits to the software development process are gained by taking this approach. Noting a strong reluctance of owners to incorporate solid software engineering practices, the thesis concludes with an outline for an experiment which goes about proving the reduction in risk at architecture exceeds the cost of that development.

ACKNOWLEDGEMENTS

My wife, Peggy. I owe so much to my wife whose been there to support me in every possible way. I have no idea why she has stuck with me during the last fifteen years, but I am ever so grateful that she saw something very few others ever did.

I owe much to Dr. Shawn Bohner, who not only guided me through this Master's Degree, but kept me funded through graduate school, as well. I thank my other committee members, Drs. Sean Arthur (excellent advice on my thesis and research) and Deborah Tatar whose education and guidance helped immensely, not only through this Master's work, but during other critical milestone's in my education. I also have to thank family members whose love, understanding and support aided my education significantly, notably, my daughter, Kayla and father, David.

The last individual I feel compelled to mention by name is the late Dr. George Gorsline, whose memorial scholarship for "most improved student in computer science" enabled me to finish my undergraduate degree on a fulltime basis and continue in graduate work. Dr. Gorsline represents a group of people I thank and acknowledge profusely: *those who ever gave someone a second, or third, or fourth chance despite all apparent evidence that individual did not deserve it.*

I gratefully acknowledge all Hokies who pulled together following the events of April 16, 2007 – your spirit of "Ut Prosim" helped inspired me to finish this thesis. In the words of Dr. Nikki Giovanni, "we will prevail... we are Virgnia Tech."

Also, to anyone who has designated themselves as an organ and tissue donor, I thank you. What for you takes so little, means literally *everything* to others. For more information, visit `http://www.organdonor.gov/donor/index.htm`.

And of course, thank you God.

TABLE OF CONTENTS

1 Introduction and Problem Statement 2

1.1 The Software Crisis 2

1.2 Reasons for Overrun 6

1.3 Risk 8

1.4 Cost of Failure 10

1.5 An Architecture-Centric Approach 11

1.6 Overview of Remaining Chapters 12

2 Software Architecture Overview 13

2.1 Introduction 13

2.2 Terms and Concepts 15

2.3 Impact of Requirements 19

2.4 Examples of Architectural Approaches 21

2.4.1 Pipe and Filter Model 21

2.4.2 Layers 22

2.4.3 Blackboard 22

2.4.4 Client-Server 23

2.4.5 Framework 23

2.4.6 Others 24

2.5 Describing Architecture 24

2.5.1 Unified Modeling Language 24

2.5.2 Other Architecture Description Languages 26

2.5.3 Where the Architecture Ends 26

3 The Architecture Centric Approach **29**

3.1 Past Research 29

3.2 Architecture-Centric Project Management 30

3.2.1 The Architecture Team 31

3.2.2 Functionality-based Design 32

3.3 Modeling the Software Architecture: The Four Views 33

3.3.1 Evaluation of Software Architecture: ATAM 35

3.3.2 Schedules and Software Development Plan 38

3.4 Transition to Low Level Design 39

4 Estimation and Risk **40**

4.1 Estimating Size 40

4.1.1 Units of Software Size 41

4.2 Estimating Effort and Cost 44

4.2.1 Estimation By Analogy 45

4.2.2 Parametric Models and Methods 47

4.3 Risk Analysis 51

4.3.1 Expected Values 52

4.3.2 Decision Trees 54

4.4 Conclusion 57

5 Addressing Risk with Software Architecture **58**

5.1 Addressing Estimation Risks 59

5.2 Addressing Quality Risks 64

5.2.1 Research Toward Formalizing Quality Requirements at the Architecture Level 65

5.3 Other Benefits of the Architecture-Centric Approach 65

5.3.1 Stakeholder Communication 65

5.3.2 Resource Assignment 66

5.3.3 The Project Schedule 67

5.4 Ending Thoughts 67

5.4.1 Why Architecture? 67
5.4.2 Conclusion 68

6 Conclusions and Future Work **70**
6.1 Observations 70
6.2 Conclusions 71
6.3 Future Work 72
6.3.1 Proving the Benefits of Architecture-Centric Project Management 72
6.3.2 Using Architecture-Centric Estimation to Define Architecture 75
References 76
Vita 81

LIST OF FIGURES

1.1 Inflection Point in Risk Curve Around Architecture 3

2.1 Waterfall life-cycle Model [Bal06] 14

2.2 Design Phase with Architecture 16

2.3 Design Phase Activities and Deliverables 28

4.1 Decision Tree for LSPRU Architecture 55

5.1 Error Associated with Software Estimation [MIT02] 61

5.2 Cost of Change Over Life-Cycle [Met05] 63

LIST OF TABLES

1.1 Molokken's Summary of Estimation Surveys 5
1.2 Software Project Outcomes by Size of Project 6

4.1 SPR's multipart taxonomy of function points 46
4.2 COCOMO II process exponent scale factors [BCH+95] 48
4.3 COCOMO II process exponent values and criteria 50
4.4 COCOMO II effort adjustment scale cost drivers [BCH+95] 50
4.5 Estimated outcomes of project at LSPRU 53
4.6 Expected Values of LSPRU Alternatives 55

Chapter 1

Introduction and Problem Statement

Projects routinely fail to complete on time, on budget and on spec. Many of these project failures are due to insufficient knowledge present at the time estimates are made. This lack of knowledge present is reflected in the high risk present in those projects. This work presents a premise that by taking an architecture-centric approach to project management, a significant amount of risk can be eliminated from the project at a higher rate than at the low-level design and implementation phases which follow. We graphically represent this in Figure 1.1. The research approach presented here examines size and effort estimation techniques in order to discover architecture factors which suggest this assertion is true.

1.1 The Software Crisis

Research continues to support the notion that a "software crisis" still exists. Software projects continue to run behind schedule and over budget, as has been reported for years [Har06]. Companies invest a higher and higher percentage of revenues into IT development [CR99, Cha05], as they incorporate automation into more of their operations. That is, an increasing amount of money is being put into IT development, but companies are still routinely not achieving good value. Systems cannot be counted on to finish on time or within budget, implement all of the required features or

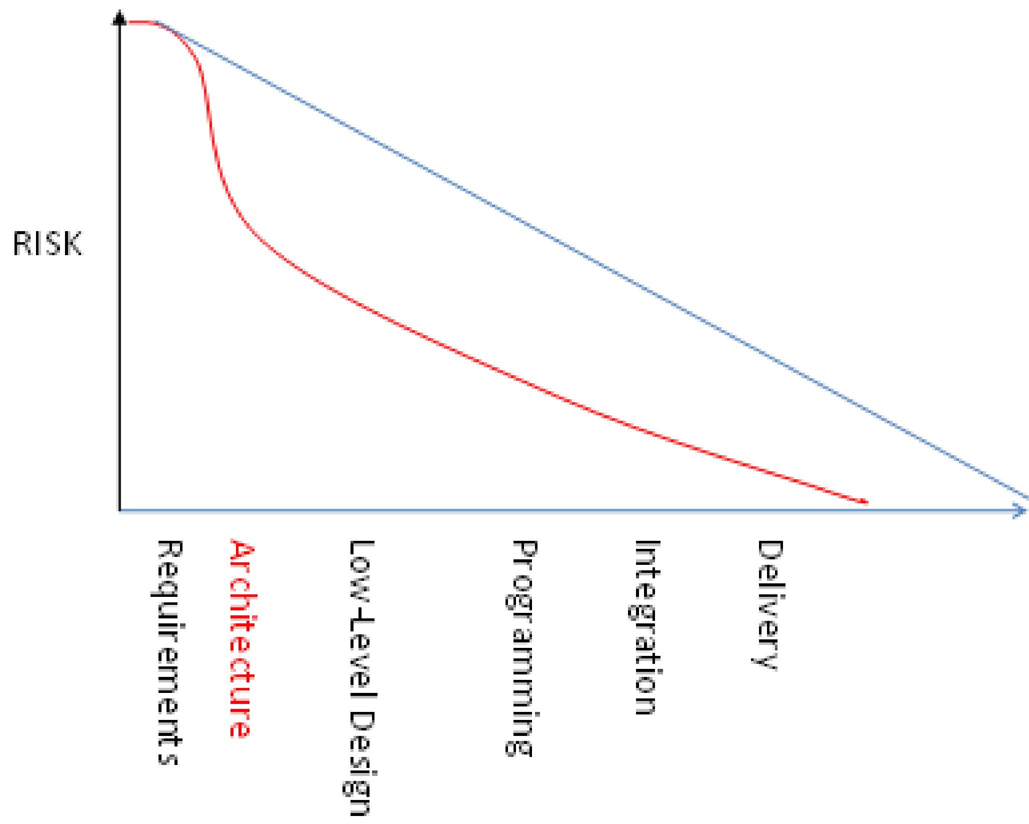

Figure 1.1: Inflection Point in Risk Curve Around Architecture

deliver the necessary quality requirements, despite sophisticated effort estimation techniques and highly mature software engineering processes. The best estimates compiled on large scale projects continue to fall far short of actual development results.

The CHAOS report by the Standish group shows that software projects experience an average schedule overrun of 84% and with an average cost overrun of 56%. Their work suggests that these numbers get worse as the project size increases. This report has been compiled from a database of over $40,000$ projects from over 3000 companies. The research of the Standish group focusses on large scale projects from companies with well defined processes. The Standish group grew out of an IBM effort to track waste in software sales. The initial study showed that decreasing sales were tied directly to a software project cancelation rate near 40%. Focus groups were then formed to investigate the reasons for project failure. The goal of the current Standish Group is to determine the factors which tend to lead to software failure, and conversely, success [CHA94].

The Standish Report breaks down the software projects in their database into three project resolution categories: success, failure and challenged. Jim Johnson, Standish Group chairman, defines success of a software project as being a project which is delivered within schedule and on budget with full intended functionality. A *challenged* project is one that may not meet one or more of the criteria for success, however still possesses significant value. A *failed* project is one which likely was never completed, or completed with a significant overrun or fewer features and is likely never released. The 2004 CHAOS report shows 53% of projects as challenged and 18% as failed, this last percentage one that Johnson cites participating managers reporting as "optimistic" [CHA94]. The average schedule overrun of 84% in particular is interesting because it shows the deficiencies present in the current models for used to estimate the development life cycle.

The numbers of the Standish group, while widely cited, are called into question by many, including Robert Glass [Gla06] who suggests the groups methodologies are lacking. Johnson of the Standish group stands by the relevance of the 1994 figures, and suggests that the primary factor behind the success rate improvement is a decrease in project size, supporting Jones' 1995 study. Johnson's claims are challenged by many [MJ03, Gla06], however, the industry still has a substantial amount of cost and schedule overruns. While other research suggests that while the Standish numbers may be higher than observed elsewhere, there is strong evidence to suggest that software projects are indeed routinely failing to meet goals. Moløkken and Jørgensen [MJ03] report that $30-40\%$ of large scale software projects experience cost overruns. While their summary of software

estimation surveys reports numbers not as critical as those of the CHAOS report, they do indicate a persistent trend of overrun. A summary of Moløkken and Jørgensen's findings can be found in Table 1.1. Note that these Standish report figures cited in are from somewhat different than the 1994 edition of the CHAOS report, which only reported 365 respondents.

All of these surveys depended on project manager responses or voluntary participation in research. Results may be skewed by a number of factors, namely the subjectiveness of the criteria and a possibly unrepresentative sample set due to an organization's unwillingness to disclose its "failures" [Gla06]. These survey results display widely different numbers due to a multitude of variables. Among these are the survey size, population, time frame and scope of the project. Each researcher also used different surveys, with different focus for the questions. Despite all of these variables, however, a persistent trend emerges whereas most software managers agree that a significant number of projects experience cost and schedule overrun.

Study	Jenkins	Phan	Heenstra	Lederer	Bergeron	Standish
Percent of cost overrun	34%	33%			33%	89%
Percent of projects over budget	61%		70%	63%		84%
Percent of schedule overrun	22%					
Percent of projects over schedule	65%		80%			84%

Table 1.1: Molokken's Summary of Estimation Surveys

Furthermore, there seems to exist a high negative correlation between software project success rates and size of the project. T. Capers Jones published research in 1995 based on a survey conducted by a team at Software Productivity Research (`http://www.spr.com/`) of six distinct categories (by domain) of large software projects [Jon98]. These domains included systems software, military software, information systems, outsourced, commercial and end-user. They used function points to measure the size of the projects and defined failure as cancelation of the project, schedule overruns above a given threshold or cost overruns above a defined threshold. Success was defined as the system on time or early, within budget and providing high quality levels and user satisfaction. The research by Jones et al. at Software Productivity Research also found a relationship between software project management factors (eg. estimation techniques, maturity of development process, experience of team) and project outcomes. This relationship is explored in Chapter 4.

Outcome	< 100 FP	$100 - 1,000$ FP	$1,000 - 5,000$ FP	$> 5,000$ FP
Canceled	3	7	13	24
Late by > 12 months	1	10	12	18
Late by > 6 months	9	24	35	37
Approximately on time	72	53	37	20
Earlier than expected	15	6	3	1

Table 1.2: Software Project Outcomes by Size of Project

1.2 Reasons for Overrun

The primary reason cited by mangers for cost and schedule overrun is requirements creep [MJ03]. That is, requirements continue to be added onto the project after implementation have begun. Reasons for this are plentiful, including unforseen scenarios, advances in technology, change of stakeholders or business changes such as the release of similar software by another company[Cha05]. All of these reasons for requirement creep can be considered a lack of knowledge. This volatility of requirements is considered a significant factor in effort estimations [Roy98] and most models make some attempt to account for this [Jon98]. Implementing new features at a later stage in the development cycle takes significantly longer than if planned from project inception [Bro95, Cha05].

Brooks suggest that optimism is an underlying factor to schedule overrun, claiming all developers are optimists when estimating project development time. The underlying assumption, he says, is that "each task will only take as long as it 'ought' to take" [Bro95]. Moløkken and Jørgensen also cite studies which still claim optimism as a large contributing factor to schedule overrun [MJ03]. This factor continues to play a role in modern estimation models, as the majority of these models, at some point, involves a subjective comparison with past similar projects. The pervading attitude of optimism in the design team can have an outstanding effect on how these comparisons are made. However, chances of project success are built on conditional probabilities, related to the dependencies among software components. The probability of the project success is a function of the dependant probabilities of each components success. The extent of this probabilistic dependency depends on the nature of component dependency. This probability is computed as a product of each components success, and the success of the interface. Consequently, the probability of meeting a deadline for the system decreases with the number of components [Bro95]. In conclusion, the notion of optimism

being cited as a driving factor of overrun is not only a psychological factor, but a failure in the design stages to anticipate problem areas. In other words, a lack of information, or uncertainty in the software development process, leads to over-optimism.

Barry Boehm, claims that up to fifty percent of the software life-cycle is dedicated to rework on large projects[BP88a]. Components and interfaces that were not properly designed for the system cause engineers to devote large amounts of time to patching interfaces and rewriting code. Inadequately defined requirements which lead to a mismatch of stakeholder expectations can also be a large contributing factor towards rework. Research notes that rework costs tend to follow a Pareto [VS94] distribution, that is, about 80% of the rework costs are associated with 20% of the problems. Boehm goes on to cite further research suggesting that this rework, when taking place early in the life-cycle, can reduce the time needed from fifty to two hundred times. These related findings give added emphasis to the need of detecting these issues early [BP88b].

Another cause of overrun is failing to discover that a particular quality requirement has not been met until after the system is built. Again, note that this reworking of the system frequently results in a cost "orders of magnitude higher than performing the evaluation and transformation of the system design early in the development cycle" [Bos00]. Early indications of all necessary features of a component gives not only better input to estimation models but lowers or even removes the time spent reworking code that had already been developed. This can oftentimes lead to a significant amount of retesting, causing even more significant delays on the system. Determining the failure of a system to adequately meet a quality requirement leads to an iterative process of requirements refinement and change. If a change occurs involving a component with high coupling, this retesting often involves a large number of system components.

Brooks succinctly discusses the compounding effects of component delays on the entire schedule [Bro95]. While some components can be developed simultaneously, there are often components whose progress cannot proceed past a point until another component is fully developed. These component delays necessarily delay the schedule for testing the integration among different components. It is at these interfaces that most testing experts agree extensive scrutiny is required [Som04]. Research has shown that the complexity surrounding component integration is a function of their coupling. The more points at which software packages interface, as well as the nature of the interface, the more potential for issues to arise, which in turn, requires more testing. This type of complexity is generally weighed heavily in formal estimation techniques [BP88b].

This level of complexity required can be measured and calculated as a function of the cohesiveness of the architecture. The industry currently possesses a number of the tools required to efficiently automate this process. Included among these are widely agreed upon languages for describing architecture (eg. UML, ADLs), metrics for evaluating the architecture, and the ability to store and analyze project life-cycle data for later comparisons. Some research has also been done in automating the evaluation process for UML metrics, which could be used in the future as a factor for estimating the risks associated with a given architectural choice. This issue is explored more fully in Chapter 6.

Selection, motivation and management of the people involved in the process is another factor Boehm identifies as a key indicator in effort estimation [BP88b, Roy98]. This property is highly reflected in cost estimating tools, such as COCOMO II and others as a significant contributor to effort estimations. Beyond the experience and training of the development team, the composition of the team itself, or people factors, can have a significant impact on the outcome of the project. This issue of forming development teams by examining team composition and background continues to be an area of ongoing research at Virginia Tech [Alk06].

1.3 Risk

Robert Charette draws on his experience in software project management, mostly with government (specifically Department of Defense) projects over ten years. He attempts to provide project managers with a tool to aid in the decision making process, by providing a systematic approach to estimating and comparing alternatives and their associated risks. This section provides a brief introduction to this concept and explores the issue of risk estimation and evaluation more in Section 4.3. Charette gives three criteria necessary for an event to be considered a risk:

- A loss associated with it
- Uncertainty or chance involved
- Some choice involved[Cha89]

Almost all elements of a large[1] software project fit these criteria, and pose some degree of risk to the project as a whole. The uncertainty present in any undertaking with software derives from a number of factors discussed previously. Notably, inherent in software is a large number of distinct components, and a nonlinear number of relationships between components [Bro95, Cha89] leading to a level of complexity not present in most engineering applications [Som04]. On typical large scale software projects, no project engineer has knowledge of the entire system [Roy98]. This requires effective communication between participants an introduces a possibility of error through miscommunication [Cha89]. The uncertainty associated with any software project is also a function of underlying assumptions on the dependant technologies. This may include unforseen constraints on the hardware, operating systems, language and compiler or other software components. Issues resulting from unforseen assumptions introduce a degree of risk in any software system.

The very nature of software gives us extensive choices of implementation strategies. Each of these choices generally presents a tradeoff for the team engineers. A certain architecture may simplify and reduce effort in implementing networked applications, while introducing a security issues that will take additional effort to address. A certain programming language may provide convenient built in data structures simplifying certain components but offer a great sacrifice in portability. A given sorting algorithm may give significantly improved performance in the average case, but perform far below others in the worst case. These tradeoffs lead to choices for software engineers at every point in the life-cycle. Inherent with every software choice is its own degree of risk. Some alternatives are well known solutions, which the engineers involved have used extensively before, while other alternatives represent new technologies to the design team. That is, each choice for the event has a different degree of uncertainty involved with it.

Charette's research involved applying formal risk analysis techniques to large software projects. This analysis method has three steps associated with it:

1. Identification of potential risk events through a structured and consistent method
2. Estimate the magnitude of each risk and its consequences, and the creation of options
3. Evaluate the consequences of risk [Cha89]

[1] The terms "large" and "large-scale" occurs throughout this work to describe projects. While no attempt is made to precisely define this, it is assumed that the projects in question involve a development cycle measured in years, a development team consisting of twenty or more engineers, and an effort measured in hundreds of staff-months.

Charette models each risk as an ordered triplet representing the scenario, its associated likelihood and a damage index [Cha89]. He presents strategies for identifying each of these values. Risk identification techniques describe involvement from each individual stakeholder, and includes events from non-technical environments, including legal, political and economic. Identification of risks continues through categorizations for risks, based on qualities such as predictability or source of the risk. Estimating the the magnitude of the risk involves computing the likelihood of an event as well as its consequence. The likelihood of an event is the probability of it occurring. For purposes of risk estimation, this likelihood will be dependant on a certain choice of action during the development process. Both qualitative and quantitative methods exist for these estimations, but at some level the probability is determined by an estimate based on the best information and experience available. Rating the sources and determining confidence in the information can also factor into risk estimation.

Evaluating risk begins with determining a level of acceptable risk. This referent level is established for each category of risks and gives stakeholders something to which compare an option's associated risks. Referent levels frequently change throughout the life-cycle of a project. That is, more risk is likely to be tolerated in early stages, than shortly before release. Evaluation continues by determining the magnitude of the losses associated with each risk and comparing this to the referent levels. Royce and Charette both suggest that "soft" approaches to risk evaluation, based on qualitative results from stakeholder input is the way most participants in the decision making process work [Roy98, Cha89]. There does exist a number of "hard" or statistically based quantitative risk evaluation techniques which researchers have applied to software. This involves models such as iso-risk contour maps which graph the probability of failure against the consequence of failure as decision tool[Cha89]. Other models such as decision trees graphically represent alternative courses of action and their effect on each other. Cost estimation tools such as COCOMO are also used to estimate risk in a project. Issues of risk analysis are explored more fully in Chapter 4.

1.4 Cost of Failure

Enormous amounts of literature has been dedicated to documenting exceptional failures of software. These include bugs in mission critical or medical systems which have resulted in loss of life, not to mention equipment and engineering hours. Often, these anecdotes provide motivation for software engineering literature to discuss the importance of solid design and testing techniques. Failures

due to deficiencies in the software after release, as well as failures associated with overrun can have significant detrimental impacts on a company. The cost of failure is also seen in the economic and social impacts of projects running behind schedule. Companies compete with each other over "first to market" releases, as well as copyrights and patents. All of these goals are compromised by significant schedule overrun, failure to implement a complete feature set or failure to meet a (sometimes unspecified) quality requirement. These effects from overrun can also result in economic, social, political loss. Forrester, an independent technology and research company, estimates a global IT budget of $2.02 trillion in 2006 [Bar06]. If, as reported by Boehm that a possible 50% of this money was dedicated to rework, this rework cost companies worldwide over a trillion dollars last year. If only 10% of the global information technology budget is wasted, the industry is still absorbing over $200 billion dollars of lost revenue. These staggering numbers suggest that a proven way to reduce the risk associated with these projects would have merit.

1.5 An Architecture-Centric Approach

As discussed, software engineering projects seem to not perform as well as other engineering disciplines in terms of delivering a successful project on time, on budget and on specification. While having explored the causes of failure associated with software, researchers have examined processes and techniques practiced by other disciplines [Bos00, Pau02, Som04]. Chief among these practices is a solid methodology for architecture design. That is, a mature process for designing the top level components of a given system. This enables not only the engineers to have a clear picture of the structure of the project, but provides all stakeholders with a common reference point early in the design process. This reference point supplies engineers, managers, and administrators with a top level view of the system, allowing for better estimation and project planning. Software engineering research has looked to adapt this architecture approach to improve software project management

Effort estimates made early in the life-cycle have been shown to be very inaccurate [Boe84, Jon98]. The architecture-centric approach suggests that a development of the architecture should take place prior to formation of estimates, product and personal schedules or work breakdown structures [Pau02]. While one can intuitively note that the further along in the life-cycle a project progresses, that less risk will remain [Cha89], academic research and industrial experience has been documented which suggests that the specification of the software architecture removes a significant

amount of this risk [Pau02, SNH95]. Because addressing quality concerns, such as performance and maintainability, prior to estimation, the effect on overrun can be constrained. Early planning for maintainability in particular can lessen the effects of requirements creep, one of the driving factors in overrun.

1.6 Overview of Remaining Chapters

Chapter 2 presents an outline of the process of developing the software architecture. Specifically, the chapter examines the phases of architecture design, the milestones of this process and the deliverables at those milestones. Following this, this work considers the reasons for further exploring the ability to significantly reduce risk through a solid architecture in Chapter 3. Chapter 4 presents an overview of techniques available to evaluate the risks associated with a software project. The chapter also explores methods of estimating the size and cost of software projects, what inputs are required for these methods, and what indicators exist to indicate their accuracy.

Chapter 5 presents specific risks which are alleviated through the process of developing the software architecture. This is done by exploring the points in the development process that experience and research suggests a given risk can be significantly more tightly bounded. Chapter 6 provides a summary of the issues surrounding widespread implementation of an architecture centric approach. This is shown in contrast to what appears to be overwhelming evidence that this approach has great benefits. The chapter also considers what further benefits might be gained from an architecture-centric approach. Further areas of research in this area are explored in Section 6.3. Lastly, the work provides ideas for how to prove the benefits of an architecture centric approach through an industrial study.

Chapter 2

Software Architecture Overview

This chapter presents an basic overview of software architecture. Section 2.1 offers a definition of this phase of the life-cycle, and discussion of the role of architecture in the software life-cycle. Section 2.2 defines commonly used terms throughout this work, specifically in discussion of software architecture. Later sections discuss architectural approaches and provide a discussion of architectural description language. No topic discussion is complete without examining the boundaries, so Section 2.5.3 concludes with a discussion of where to distinguish between architecture and low-level design.

2.1 Introduction

Software Architecture takes on different meanings throughout the field of software engineering, and even more relevant to this work, within industry [Bos00]. While experts in the field take differing ideas on what exactly comprises the architecture of a software system, this work uses Bass' frequently cited definition of "the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships between them" [BCK98]. That is, the software architecture gives us a first slice at the design of the system, providing a top level view of major components and how those components inter-operate. In the software life-cycle waterfall model shown in Figure 2.1, the architecting of the system is seen to occur immediately following the requirements engineering. The end result of a well designed software architecture is to achieve *conceptual integrity* [Bro95] which leads to ease of use, or in the case of large software

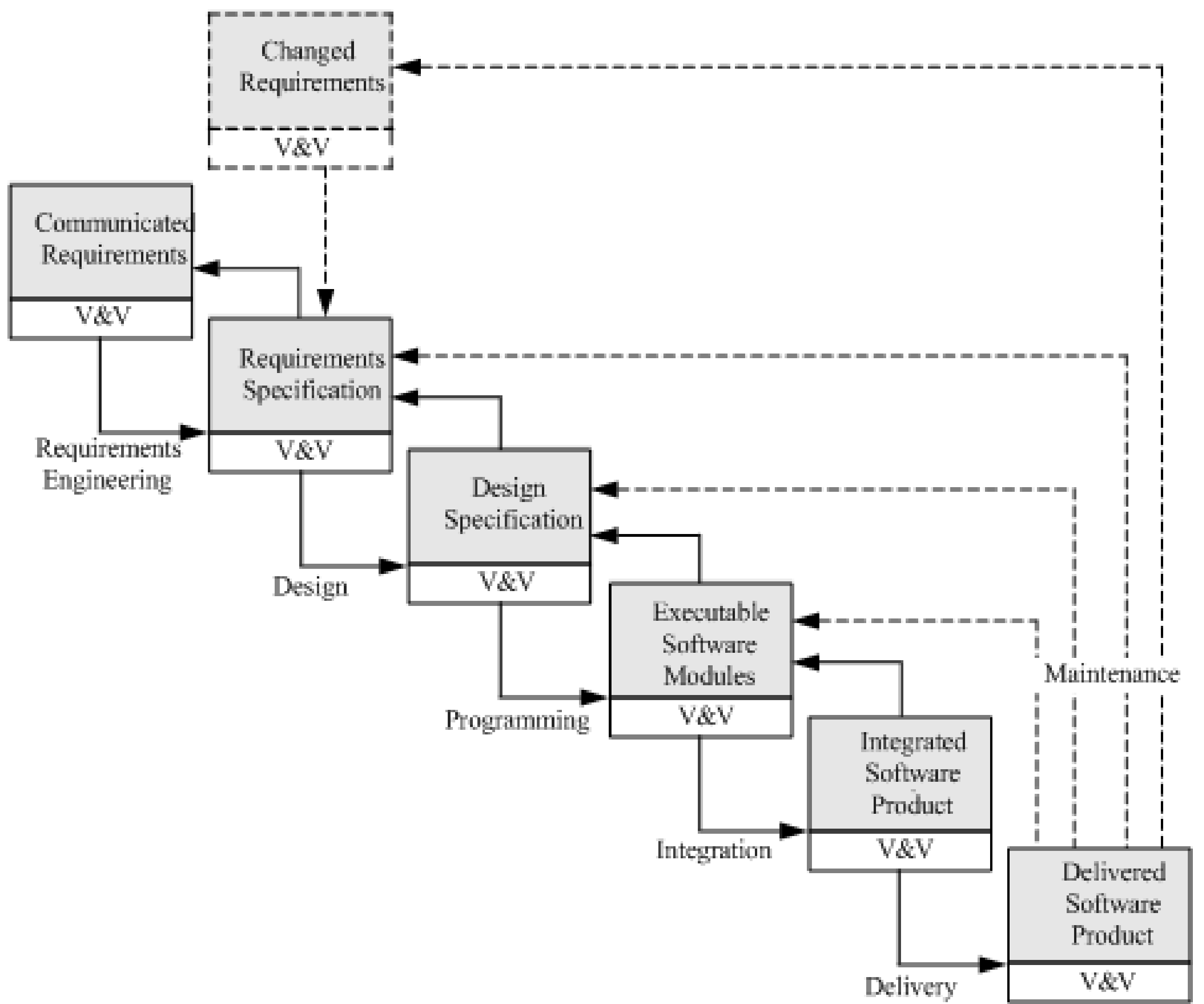

Figure 2.1: Waterfall life-cycle Model [Bal06]

system development, ease of implementation. The level of detail necessary to achieve this design, again, is interpreted differently by software engineers.

The ambiguity over what comprises a software architecture lies not only in the level of detail provided, but how the architecture is modeled. Baragry and Reed [BR98] researched the difficulty the software engineering community is having in building a consensus definition of "software architecture." This lack of consensus still exists today, however, it is no longer viewed as ambiguity, but rather as a tool necessary to project management [FBD06]. With regard to representation of the architecture, research suggests that different models possess different characteristics which in turn describe unique components of the system. This opinion is demonstrated in Paulish's "Four Views of Software Architecture" which is explored in Section 3.3 [Pau02]. Some models and description languages describe components very well, but provide little description of the connections between those components. Others represent the relationships between requirements and components to highly varying degrees. Different methods of describing architecture are explored in Section 2.5 Consideration of how industry differences in a definition of architecture in these respects may be not only desirable, but necessary is addressed in Section 5.3. This view of architecting as a diagonal slice across the design process is illustrated in Figure 2.2.

The software architecture serves as an early indicator of the major components intending to comprise the final deliverable product. The choices made in determining these components are the first attempt by project stakeholders in balancing the tradeoffs implicit in the requirements [GB98]. The architecture, once designed, gives all stakeholders a high level snapshot of the system, opening a channel of communications between those various stakeholders [Som04]. As one of the primary goals of an architecture is to incorporate quality as well as functional requirements at an early stage, this property of providing a communication mechanism involving all stakeholders is of significant importance. Chapter 3 examines what benefits this added communication serves.

2.2 Terms and Concepts

functional requirements

This term is applied to the set of requirements which describe how the system should behave and which functions it should provide. These requirements address issues such as what type of input should be handled, and what features should be provided. Functional requirements define the

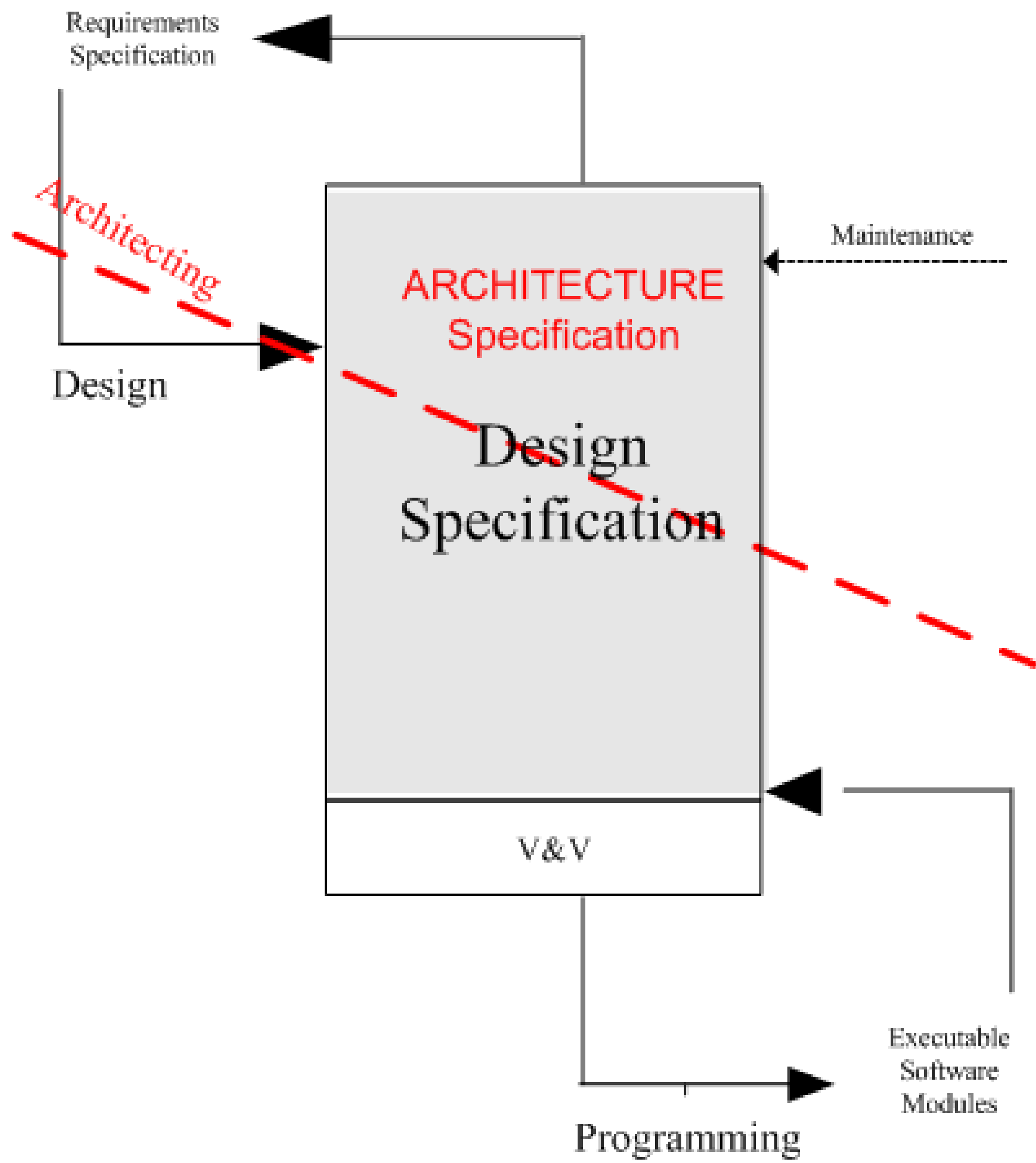

Figure 2.2: Design Phase with Architecture

boundary of the system, specifying how it should interface with other systems and users. These requirements are generally evaluated on their completeness and consistency, that is, the amount of the user requirements which are specified sufficiently, and the degree to which there does not exist contradiction between requirements. In modern functional practices, requirements are generally represented as use-cases. [Som04].

quality requirements

Also known as non-functional requirements, this set of specifications addresses issues not directly related to the functionality of the system. Quality requirements consist of properties such as *performance*, *performance*, *reliability*, *safety* and *maintainability*. There exists a considerable amount of ambiguity over the representation of these properties during requirements engineering. As in many issues in software engineering, the distinction between quality requirements and functional requirements is not always clearly delineated. *Security*, for example, could be considered in either category, depending on the specific project needs, development environment and required aspects of the system.

life-cycle

Life-cycle is a term applied to the entire process of developing a software system, usually beginning with "market," or "raw" requirements, leading to formal requirements engineering and concluding with release and maintenance. The software life-cycle is represented with different models and software engineers have not yet reached a consensus on what phases and labels for those phases should be present in a given model. This work, generally considers the software life-cycle using the waterfall model as depicted in Figure 2.1, incorporating the phases of requirements elicitation, design, unit coding, unit testing, integration, system testing and release. Validation and verification steps exist at each phase of the life-cycle, allowing stakeholders to review the artifacts at each phase, ensuring that they accurately represent the system's requirements and that the development process is correct. This model also includes feedback transitions, reflecting the iterative nature of software development.

stakeholder

The term stakeholder refers to any person who has a *stake* in the project [Mac01]. Project stakeholders include developers, managers, architects, testers, owners, end-users and maintainers.

performance

Performance refers to the system's transaction speed and throughput[CKK02]. That is, the time to

process a transaction and the number of transactions the system is able to process in a fixed time. These qualities are under a wide variety of conditions in order to assess the system's performance at peak loads and determine the average and the bounds of the speed and throughput. Performance is generally stated as a quality requirement.

safety

Safety is the ability of the system to operate without failure or cause undesirable damage or loss [Som04]. This quality requirement is usually referenced in discussion of real-time mission critical or life critical systems, although is similarly relevant in situations where system failure can result in a business or financial liability to the owners. Safety requirements may be quantitatively expressed in terms of failure rates by type, in life or mission-critical systems, a failure rate at or near 0 is the only acceptable level.

availability

Availability is the "readiness of a system to deliver services when requested [Som04]." This requirement is usually specified in terms of percentages indicating the percent of time that specific services (or the system as a whole) is able to perform specified functions.

maintainability

Maintainability is the degree to which the system can easily be modified as bugs are found and requirements change.

subsetability

Subsetability is "the ability to support the production of a subset of the system. [CKK02]" That is, this reflects the quality that a useful product can be delivered should the schedule slip and not all features be implemented. This quality is implicity addressed through some processes such as agile development.

architectural approaches

This thesis uses the term "approaches" to group together what are sometimes referred to as *architectural patterns* or *styles*. This refers to a generic model for representing architectural designs. Much of architecture literature makes subtle distinctions between these terms, while simultaneously stating the caveat that no clear delineation exists where one leaves off and another begins. Because the focus of this work is not on the development of architecture, it uses the term "approaches", favored by Carnegie Mellon's Software Engineering Institute (`http://www.sei.cmu.edu/`), to encompass all of these concepts [CKK02]. Styles and patterns are frequently seen as a solution template for a

specific class of problems. Common styles include the Client-Server Model, the Distributed Object Model and the Layered Model. Each of these styles comes with a specific set of tradeoffs in terms of ease of implementation, efficiency, portability and other qualities. Example styles are presented in Section 2.4.

domain

The term refers to the larger "category" or set to which a software system belongs. These are sometimes considered as either business domains (eg. defense intelligence project, medical system, business data mining) or computer science domains (embedded real-time, operating systems, desktop application). Section 2.5.3 discusses how differing domains tend to impose different requirements on an architecture.

archetype is the core abstraction on which the software system is structured [Bos00]. Software archetypes represent a high level abstraction of the architecture. These components differ from subsystems in that an archetype describes a unit of abstract functionality which is often reused in many places within the system. A subsystem, on the other hand, describes a subset of system functionality. Examples of archetypes include "controller,", "device," and "handler." Archetypes are often represented as class or package diagrams, describing the highest level of the inheritance or aggregation hierarchies.

architectural description languages

An architectural description language (ADL) is a representation of the architecture in a form that conveys meaning to some group of stakeholders. The most common format today for modeling a software architecture is the Unified Modeling Language (UML) [FDH06, Kru02]. Several other ADLs exist, providing alternative representations of software architectures. See Section 2.5 for a more detailed discussion on this topic.

2.3 Impact of Requirements

It is well understood that the layers of the waterfall model do not accurately reflect the obscurity between phases of the life-cycle. Sommerville remarks that "a system specification should not include any design information. In practice, this is unrealistic except for very small systems" [Som04]. The transition from requirements engineering to architecture may present one of the starkest examples of this blending of life-cycle phases.

The phenomena of "requirements creep" or "feature creep" is so well recognized in project management, that most estimation tools make some account of requirements growth in their formulas [Jon95]. This notion of a blurring of the distinction between phases is also captured in the life-cycle model's feedback loops, representing an iterative process between each phases. The profound impact of changed requirements is graphically illustrated in Figure 2.1.

In addition to the inherent iterative nature of the life-cycle, the quality of the requirement deliverables can have a large impact on the development of the system architecture. Requirements engineers, like software architects, tend to differ on where requirements elicitation and analysis leaves off, and design begins. More recently, the practice of capabilities engineering has emerged as a process to bridge this gap. This practices identifies sets of capabilities at the requirements stage that are possess change tolerant characteristics, minimizing the impact of volatility [RAB07]. Further, the more formally the requirements are expressed, generally in terms of use-cases, the higher the levels of quality. Quality of requirements is generally measured in terms of *traceability*, *consistency* and *completeness*. These are the properties of being able to map the requirement to the design components through requirements documentation, the degree to which requirements do not possess conflicts, and the degree to which requirements are unambiguous and that all services required by the stakeholders are defined [Som04]. The extent to which the requirements possess these capabilities will generally be reflected in the effort necessary to produce a solid architecture [Pau02].

Furthermore, requirements often contain a subset referred to as "business domain requirements." These requirements commonly dictate a design decision to the architecture team, based on business specific needs (be they perceived or actual). For example, a business currently implementing all their distributed applications in J2EE framework(`http://java.sun.com/javaee/`) might have concerns about maintainability and extendability of a new server side system. In the pure waterfall model, these concerns would be expressed as non-functional requirements and converted to use-cases specifying those qualities. The system architects would then have a degree of latitude to exercise in how to meet those requirements. It is common, however, for the requirements documents to specify that the same J2EE framework be used for the new system. This in effect, constrains the design and causes the requirements engineering output to contain decisions normally made in the architecture stage.

2.4 Examples of Architectural Approaches

This thesis uses the generic term *approaches*, favored by Carnegie Mellon's Software Engineering Institute, to describe what are often termed "styles," "patterns," or "models." While much of software engineering and architecture literature makes distinctions between these terms, there is little consensus as to precise definitions and all agree that no distinct line exists as to where a "pattern" departs from a "style" or from a "model" [Som04, Bos00, CKK02]. Debate over this issue is left to others. Rather, the examples presented in this section provide the reader some knowledge of commonly used architecture approaches and some of the inherent tradeoffs that occur when choosing among them.

Architectural approaches are designed to "improve the possibilities for certain quality attributes for the system on which the style is imposed" and, consequently, tend to offer less support for other quality attributes. That is, each style presents a tradeoff to the architect. Multiple architectural approaches can be, and are commonly, used together within the same system, with attention paid to the constraints inherent in each approach. The choices made by the architecture team concerning style will have an immediate impact on the ability of the system to meet its quality requirements.

2.4.1 Pipe and Filter Model

This approach models a dataflow network where the data is transmitted through the pipes, and acted upon by the filters, "gradually transforming inputs to outputs" [Dob02]. Filters consist of self contained explicitly defined input and output interfaces. A common instance of this model is witnessed in the Unix operating system shell, which is composed of numerous small executables, each with a pre-defined and well contained function. These "filters" can then be "piped" together through shell level pipes and redirection operators.

The pipe and filter model varies in its support for performance, reliability, safety and maintainability. The model naturally lends itself to concurrency given that there do not exist too many time dependencies between processes. However, if these dependencies exist, this model potentially does not provide high performance, nor high reliability, as a single failure could bottleneck the entire system. Similarly, maintainability is supported through small, encapsulated subsystems modeled as individual filters. On the other hand, due to the normally small granularity of filters in this model, a change to a single requirement could potentially impact numerous filters [Bos00].

2.4.2 Layers

This approach decomposes the system into a set of layers representing increasing levels of abstraction. Each layer provides an interface for using this abstractions to the layer above it. This approach builds on the traditional hierarchically viewed system. Different variations of the layered approach exist, differing primarily in the quantity of layers, their roles and the types of interactions between them. Some approaches allow higher levels to communicate directly with all lower levels, while others allow communication only with the immediately lower level. Other allow bi-directional communication as well.

A result of the layering approach is that performance suffers due to the multiple layers of communication necessary between components on different layers. The high coupling between layers leads to similar concerns for reliability, as a failure at one level can impact all levels represented above it in the hierarchy. Maintainability, on the other hand, is generally viewed as a positive aspect of the layered approach due to well defined and constrained relationships between components at different layers. Safety and security are supported well by this approach, as this approach allows for monitoring functionality to be inserted into the design as their own abstract layers, requiring interaction with this layer before certain operations [Bos00, Som04].

2.4.3 Blackboard

This approach utilizes a central data repository from which other components interact. These components use this singular "blackboard" as both the source from which to draw input, as well as the default destination to store results. The interactions between components is limited to reading and writing from this central repository. While the pure blackboard model specifies a single repository, this can be an interface to a distributed repository through which the components interact. Differing implementations of this approach allow for various levels of priorities to be assigned to system components.

Due to concurrency issues and a lack of well defined control flow, this approach can present performance issues. Attempts to impose control flow on the system may improve these issues, but at the cost of increased computation. Similarly, the lack of explicitly defined control flow has large safety concerns, a failure of one component to read or write accurate data to the central repository can lead to a system wide failure. Security problems can exist if sufficient precautions to validate

data access are not taken, but the central repository simplifies control of this access.

2.4.4 Client-Server

The client-server model organizes the system as a set of services, servers which provide them, clients that use them and a network to allow access and communication. This over-simplistic description suggests that the main logic of the system resides on the servers, although this varies significantly in practice. In reality, both thin-client (most of the data processing and management resides on the server) and fat-client approaches are used. The client-server approach works by allowing clients to make requests through a defined protocol which are then handled by the servers. Clients must be aware of the servers, but are not required to have any knowledge of other clients.

The client-server model works well to optimize performance over a distributed system, however the performance of the network itself provides a performance limit often more constrained than any individual component. Processing can be distributed easily among multiple processors, allowing for high concurrency. This model presents inherent tradeoffs in reliability and maintainability. These qualities are supported by little dependence on individual clients, but are subject to failure of the network and the servers. Systems employing redundant servers can be upgraded transparently to clients. The issues around data security and safety are similar to those found in the blackboard models. The popularity of this approach has led to significant research in multi-tiered approaches to improve qualities such as security and reliability.

2.4.5 Framework

Framework, or computational structure of a system, provides an infrastructure on which lower level components are then implemented. In many cases, the framework also provides a reusable design for a system, and in practice, a number of frameworks have been designed specifically to support reuse. The purpose of the framework is to allow developers to expend more effort on implementing system functionality by capturing the lower level details of the system in a reusable and extendible package. The framework usually dictates where functionality will reside in the system, and which

Framework represents a finer grained specification of the architecture than the approaches described above [Bos97]. As noted previously, in industry it is common experience that with a complete set of requirements is often included specifications concerning which framework the architecture shall be implemented. From this work managers are frequently able to remove a substantial amount of

risk from the project. Evidence suggests that effort estimations can likely significantly improve precision by considering framework as part of the architecture [Jon98].

2.4.6 Others

There are far too many architectural approaches to describe in this work. Some of the more popular among these are push-based, peer-to-peer, event based and call-return architectures. Again, the distinction between which of these approaches are best classified as models, styles or patterns is not of immediate concern. It is sufficient to note that each approach to designing an architecture possess a set of tradeoffs to the architect, and that approaches are generally combined together within large systems, respecting the constraints inherent in each. Notably, once a decision on an architectural approach is made, constraints are placed on the quality attributes of the system [SC06] and translates into the amount of risk present in the project. For further discussion on several architectural approaches, the reader is referred to works of Shaw and Garland [SG96], Bass et. al [BCK98] and Bosch et al. [Bos00].

2.5 Describing Architecture

The need for a language to describe a system architecture dates back to 1975 when cite deRemer and Kron researched the difference between programming large systems and small programs [DK75]. They suggested that there would be benefits for a language to describe major components and the relationships between them at an early stage in the life-cycle. Today, numerous architecture description languages exist, all of which allow for various representations of a system's components and relationships between them. Some ADL's are incorporated into computer aided software engineering (CASE) automated systems for generating source code from architectural models. As one of the primary objectives of an ADL is to facilitate understanding of the architecture by all stakeholders, most languages incorporate multiple graphical and textual descriptions of the system.

2.5.1 Unified Modeling Language

The Unified Modeling Language (UML) remains the most prevalent ADL in industry today [Som04]. UML is officially defined by the Object Management Group (`http://www.omg.org/`), a non-profit consortium whose aim is to provide industry standards for numerous technologies. The current

version, UML 2.0, provides thirteen distinct diagrams, each of which serves to present a different view of system components. The collection of UML diagrams are augmented with documentation such as written use-cases in order to provide meaning and cohesiveness to the model. UML diagrams provide the means to add an arbitrary degree of specificity to the model, far beyond the level of detail normally considered to be part of the architecture. This allows for the same language to be used throughout the design of the software system.

UML package diagrams are frequently used to provide a snapshot of the highest layers of component groups. *Use case diagrams* provide the mapping between system requirements and system components. These diagrams show the behavior of the system with respect to actors, and provide designers with an indication of how functional requirements will be traced to classes or other components. Package diagrams provide the ability to add level of detail through dependency connections and nested packages. Component diagrams show physical components such as libraries, files and executable packages. These diagrams model the manner in which system components interact and depend upon each others. Class diagrams in UML model classes present in an object oriented system and are one of the most prevalent diagrams in a UML model. These diagrams generally allow for significantly varying degrees of fidelity, allowing system designers to fine tune the "detail knob" in order to provide stakeholders with pertinent information. Class diagrams are sometimes augmented with *composite structure diagrams* which model the internal structure of a class and how the class interacts with its boundaries [SX03].

Other UML diagrams focus on demonstrating the interactions between components more so than the structure and relationships of the components themselves. Among these are state diagrams which represent states of the systems and the transitions between them, activity diagrams which model the real world business process which the system is designed to produce. Sequence diagrams are used to show the interactions between components over time, and the messages passed between objects at specific points. New to UML 2.0 is the *timing diagram* which provides another view, similar to sequence diagrams, to visualize the changes components undergo over time.

UML was designed to be a general purpose modeling language, with aim toward supporting an object oriented design methodology. It has been criticized for not offering enough support for other methodologies, and with respect to architecture, for not providing a consistent notation for connectors [SC06]. Despite its apparent deficiencies, UML adoption remains widespread and has been incorporated in numerous automated tools and formal processes, notably IBM's Rational Unified

Process [Kru02]. Research is ongoing in an attempt to extend UML to provide more structured support as an architecture description language, notably through adding more diagrams and tighter specification of interactions [Oqu06a].

2.5.2 Other Architecture Description Languages

Oquendo proposed highly formal methods, the π-Method [Oqu04] and subsequent ı-Method [Oqu06b], to formalize the assessment of architectures through mathematical means. These ADLs were designed to offer support for modeling the changes undertaken by a system during runtime. Sashofy et al. proposed an approach to developing an ADL based on the eXtensible Markup Language (XML) in order to facilitate generation of ADL's and to leverage the extensibility of XML to adapt the language for a specific domains needs [DvdHT05]. Numerous other architectural description languages have been proposed and implemented to varying degrees, however, outside of specific domains, have failed to gain widespread acceptance.

2.5.3 Where the Architecture Ends

There currently exists no consensus of where architecture leaves off and design begins within the software engineering community. Perry and Wolf suggest that "architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for design." This differs from their notion of design which is described as "concerned with the modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and satisfy the requirements" [PW92]. This explanation begins to give us some intuition as to where the distinction is made, however, allows for a large degree of subjectivity. Daniel Paulish of Siemens Corporation suggests that architecture stops when a "module can be given to another developer for a detailed design" [Pau02]. This is analogous to the concept of design ending at the point at which it can be given to a programmer for implementation. Clearly, not every engineer involved in design and implementation will require the same level of detail to carry out their task. The junior engineer is likely to require more detail than the experienced, or senior, engineer. The ambiguity in this statement is justified by Paulish's insight that in practice, the software architects remain involved in the project past the milestone of "architecture" and that design is an iterative process whereby the architects will receive feedback

from other stakeholders (in this case, specifically developers conducting low-level design).

This degree of subjectivity present in the architecture of a system was actually found to be a necessary tool of the project manager by Fairbanks [FBD06]. Their research indicates that the granularity of design *should not* remain constant between business domains, application domains or organizations with differing structures. Specifically, this "detail knob" allows the manager to specify the level of fidelity to the architect which necessary in order to accurately asses size, effort and risk of the project without expending more effort than necessary at the architecture stage. This notion of architecture being an "adjustable" based on the domain is graphically shown in Figure 2.3. These qualities, are explored in detail in Chapter 4, aid the project manager in decision making and provide the insight necessary to produce engineering artifacts such as the work breakdown structure and project schedule with milestones. This notion of an architecture "detail knob" supports the premise that the amount of detail present in the architecture is the point at which, from a project management view, that resources can be allocated to project components.

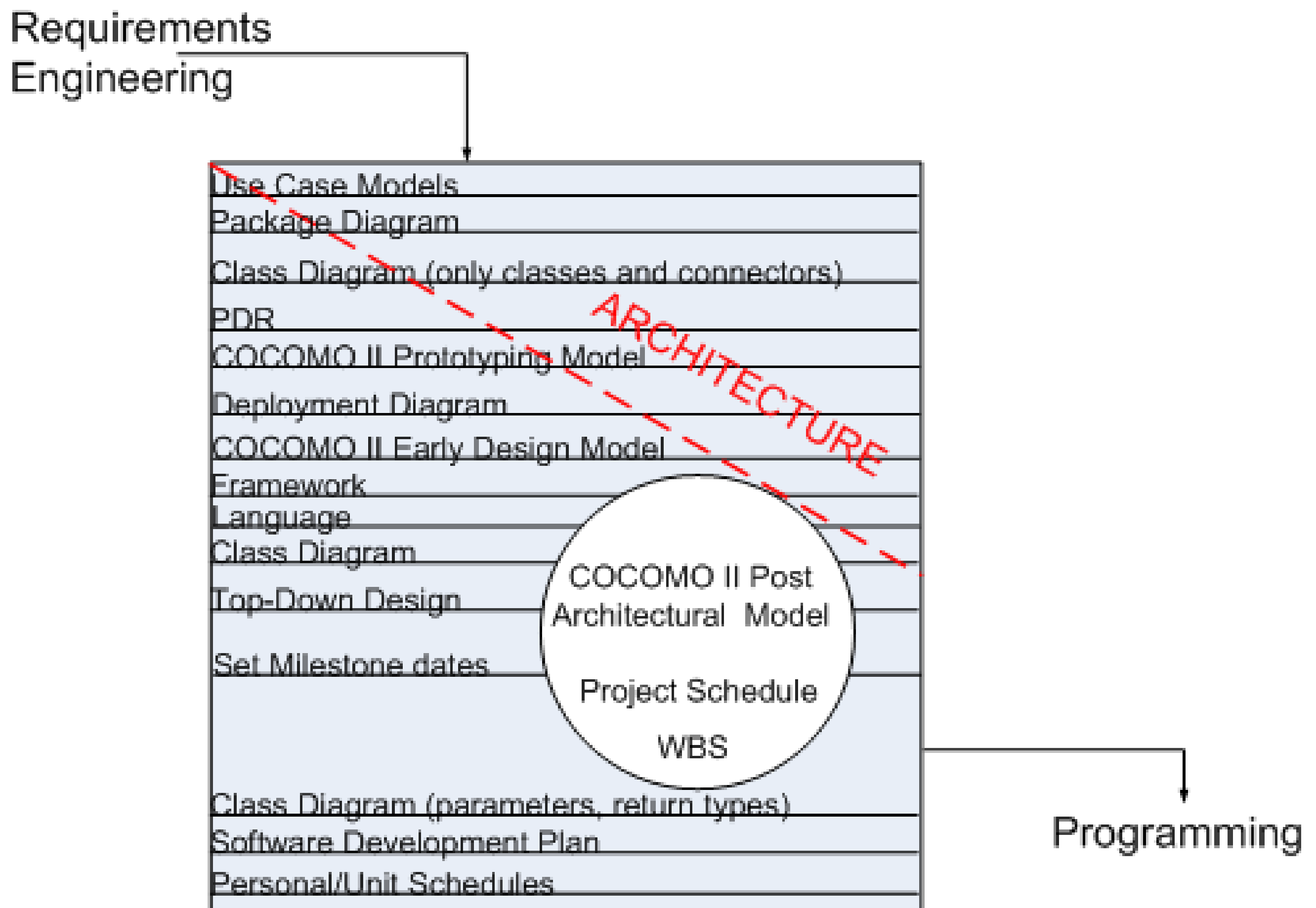

Figure 2.3: Design Phase Activities and Deliverables

Chapter 3

The Architecture Centric Approach

3.1 Past Research

Daniel Paulish at Siemens corporation published extensively on the success experienced by taking an architecture-centric approach to project management. Other Siemens architects also published results of their experience, and outlined the approach that has become known as "global analysis" [NPSS01]. The Siemens architects suggest presenting multiple representations of the architecture, in an attempt to make the design relevant to all stakeholders and to capture the essential components and their relationships. While the concept of software architecture dates back much further than the work at Siemens[1], their chief architects were among the first to publish methods, lessons learned and reports of success with this approach. Today's definitions of software architecture differ slightly, but longstanding software engineering principles related to solid and early design techniques are still present in modern architecture development and analysis methods.

The projects that researchers at Siemens draw their experience from are self-categorized as mid-size to large projects: those involving from nine to over sixty software engineers with a life-cycle of one to two years. Some of these projects involved distributed, multi-national teams, others

[1] in 1969 Fred Brooks defined architecture as the "conceptual structure of a computer ... as seen by the programmer" [Int07b]

involved outsourcing of significant components, while others still involved collaboration between Siemens and other companies. Much of the work at Siemens took place in the domain of e-Business, security related technologies, medical applications and industrial automation. These projects also involved a wide variety of computer science domains and scopes, including GUI applications, real-time embedded systems and networked systems [Pau02, HPB05].

Fairbanks et al. at Carnegie Mellon found, through involvement with projects at a large financial firm, that applying an architecture centric approach provided significant benefit to the organization [FBD06]. This research extends past Paulish's experience in that Fairbanks was exploring the application of an architecture first approach in a business whose primary area is not software. They report on the success they had implementing this style of management with four large projects, each with significantly different scope. One project was being developed from the ground up, with no legacy code involved. Two others were significant modifications to existing systems and the fourth project involved coordinating three others and providing communication mechanisms. Their research employed their own modeling technique, referred to as a "synthesis of existing published techniques" [FBD06].

Other publications relating success with architecture-centric approaches also exist. Medvidovic et al. found that this approach used in development of software development environments let to high extensibility (a primary quality attribute needed for software environments) and acceptable tradeoffs among other quality attributes [MOT+00]. These methods were only tested on smaller systems, however the authors note future work being planned to integrate architecture-centric processes into large software development environments. Nord and Tomayko recently examined how an architecture-centric framework can be used to anchor an agile development process, drawing parallels between concepts inherent in each. Far from suggesting that agile methods should lack in formalized structure, their work suggests that agile an architecture first approach allows for rapid development and iterative planned releases while ensuring the quality attributes important to stakeholders are accounted for during the process [NT06].

3.2 Architecture-Centric Project Management

Brooks claims that integrating and testing separate components into a cohesive project imposes a time factor of three on the development time, contributing to an overall increase of nine-fold.

That is, to integrate separate components into a usable, release quality system takes approximately nine times the effort required to develop those components as independent entities [Bro95]. The importance of well-designed architectures with highly specified interfaces was observed over thirty years ago, yet this style of project management is still not mainstream. All software projects are understood to have an architecture, either explicitly or implicitly defined, and there is no practical way to implement a large project without some form of high level design document. However, there remains an unwillingness on the part of management to dedicate resources to a formal architecture development method and subsequent process to identify costs, risks and tradeoffs from the architecture deliverables, instead relying on informal estimates which fail to account for many quality requirements [Jon95, Roy98, Pau02].

3.2.1 The Architecture Team

The architecture team consists of a small number of stakeholders, generally the project manager, a chief architect and a small group of high-level designers. Paulish suggest a number no more than six developers on this team for mid-sized projects [Pau02], although later research expresses this team size as a function of the number of component teams [HPB05]. This small number of stakeholders involved at this stage supports the concept of the architecture being a strictly high level design, and avoids the exponential communication complexity involved with a large team at an early stage of the life-cycle [Bro95]. Paulish warns of dangers associated with a team too large, drawing parallels to Brooks' oft cited observation that communication complexity exponentially as team size increases [Pau02, Bro95]. Fairbanks published research on the other side of this issue, citing that an architecture team which is too small fails to produce useful models at all [Fai03].

It is generally understood that the role of the architecture team will continue beyond the specification of the architecture and succeeding plans and estimates. That is, those involved early on in the high-level design will take on roles in successive development steps as well as in change maintenance to the architecture itself. To this end, Paulish recommends a representative from each component development team be a member of the system architecture team, generally the component team leader. This is based on the assumptions that a team leader is responsible for no more than seven software engineers. By incorporating a representative of each component team at the architecture stage, a sense of ownership in the entire process is fostered [Pau02].

3.2.2 Functionality-based Design

Several schools of thought exist concerning best practices and approaches to all levels of software design. As one of the chief goals of a formal architecture development process is to ensure the likelihood of meeting quality requirements, a manner of incorporating these into the design process is necessary. Bosch suggests that conventional design methods in computer science tend to under-emphasize quality requirements, although some practices, such as object-oriented design, claim to implicitly incorporate qualities such as reusability and maintainability into the system [Bos00]. The practice of functionality-based design presented here was found by Bosch and others to be an "objective and repeatable" design method which organizes the process into indistinct phases that ultimately results in a desired architecture [Bos00, Som04].

Bosch's approach to software architecture design is to begin by developing an architecture based on the functional requirements. This begins by examining the boundaries of the system (the system context): its interface with external entities. These entities may be other software systems with similar or widely different purposes, hardware components, and users themselves. The software systems comprising this context may include higher-level systems, lower-level systems and peer-level systems. Once the context has been defined, specific interfaces can then be designed and specified between the system and its boundaries. These interfaces generally fall into categories such as "uses," "used-by," and "depends on."

Requirements are then associated with each interface. Both functional and quality requirements are assigned to each interface, which in turn allows for a higher degree of specificity during refinement. By ensuring that all interfaces are associated with all relevant requirements, architects are fully aware of all stakeholder needs and considerations. This suggests less need to rework subsystems which often can often lead to overrun. Reexamining the mapping of quality requirements to system components becomes a dominant goal of architecture evaluation, but incorporating considerations of these at early stage allows the architecture team to make decisions which support quality goals deemed most important.

Next, the primary issue is to identify the archetypes on which the system is to be structured. The first iteration of the software architecture is to design a small set of abstract entities which describe the majority of the system behavior at a high level. These archetypes (Section 2.2) form the most stable part of the system in that they only change in small increments and in limited manners. Candidate archetypes are identified by noting during the architecture design process

recurring patterns of common characteristics between parts of the system. Candidate archetypes are studied and those which are duplicated or whose intersection of functionality is large are combined and refined until an appropriate set of archetypes is determined.

Following identification of the archetypes, the system is then decomposed into components which are instantiations of those archetypes. These components are primarily specified in the manner in which they interact with the system's context. These component interactions may include user interfaces, hardware interfaces, network protocols and interactions with other software systems. The inner workings of an architecture's system components (ie. algorithms, data structures) are usually reserved for succeeding, lower level, design phases. The descriptions of these components are used to verify that the architecture describes a system which meets its functional and its quality requirements, or proposes an acceptable tradeoff. These components comprising the architecture and the relationships between them are expressed in models (Section 3.3) and an iterative process of evaluation and refinement Section 3.3.1) continues until an acceptable tradeoff is reached.

3.3 Modeling the Software Architecture: The Four Views

Section 2.5 examines the need to express high level design in a standardized format capturing the components and relationships that comprise the architecture. As a significant factor in the analysis of an architecture is input from a sampling of all stakeholders, most research supports the use of multiple diagrams, tables and textual commentary to express the features in a meaningful way to all interested parties. To this end, this section presents what Paulish terms the "four views of software architecture" [Pau02]. Building on the widespread acceptance of the Unified Modeling Language as an architectural description language (see Section 2.5.1), discussion follows of how various diagrams can be used to express these four views. Information on the four views and their realization using UML is described in the published works of Siemens' research [RKJ04, HNS00].

Conceptual View

The conceptual view describes the system at a very high level, showing identified archetypes and their relationships. This view is is intended to be highly relevant to non-technical stakeholders, as it is frequently associated with the application domain of the system by identifying problems and their solutions primarily in domain terms. This view should be independent of software and hardware

techniques, tools and techniques, rather closely modeling the domain process being modeled. The conceptual view addresses the validity of the architecture, how functionality will be partitioned into system components, and provides a first indication of how changes in requirements can be minimized in the system.

The conceptual view can be visualized in UML with class diagrams showing stereotyped classes to view the static configuration. These abstract class diagrams represent the archetypes of the system identified during functionality-based design. It is understood in this view that these class diagrams represent a class of solutions more so than classes intended to be present in the final system. Sequence diagrams demonstrate interactions among components, allowing stakeholders to visualize changes in the system's archetypes during processing of a task.

Module View

The module view of an architecture shows the system's organization as a layered representation. The goal of this view is to organize system in such a way as to minimize the impact of change in system boundary elements. The module view is relevant to project managers and team leaders as it helps focus the software engineer's expertise, improving process efficiency. The module view provides a mapping between the conceptual solutions presented above, and the platforms and technologies intended for use with the system. The module view provides stakeholders with indicators for validating interface constraints and early indicators to assess configuration management.

Class diagrams are used to demonstrate module decomposition, utilizing nesting of classes through aggregation. Subsystems and represented layers shown with package diagrams, again using nesting and UML dependencies to show relationships between layers. UML class and package diagrams provide a number of dependency representations (eg. association, aggregation, inheritance, etc.) allowing for assessment of complexity. These diagrams indicate how the product will be realized in a given framework and how subsystems can be organized to support reuse.

Execution View

Class diagrams again play a role here, demonstrating the execution configuration of the system. The execution view provides insight into how a system will meet quality requirements such as performance and maintainability. This view demonstrates how the system handles runtime considerations such as memory usage and hardware assignments between multiple processors.

Deployment diagrams are used to demonstrate how the configuration corresponds to system hardware components. Sequence diagrams show the dynamic behavior of the system, especially important in network-centric applications. State and sequence diagrams illustrate the protocol of a communication path.

Code View

The code view is the lowest level view, often viewed as more a part of detailed design effort. This view maps how modules and interfaces will be mapped to source files and how runtime components in the execution view correspond to executable files, including existing legacy components and libraries. This view has particular relevance to programmers, who require specific knowledge of languages, application programmer interfaces, libraries and source code organization. The code view is especially useful for assessing performance, maintainability and portability.

Component diagrams are the UML artifact which show these components' organization and their dependencies. This view relies also on tables and some method of trace dependency in order to map elements in the module view to their respective source components.

The four views of software architecture are necessary to ensure that all stakeholders gain an understanding of the proposed system. Each view is realized with a different set of models and therefore conveys a different set of information. System designers and programmers may use the execution and code view to gain understanding of their components place in the system design and what interfaces they are concerned with. Non-technical stakeholders may use the conceptual view to gain understanding of how quality concerns and functional requirements are realized in the architecture. The ability to convey the architecture to all stakeholders plays a primary role in evaluation.

3.3.1 Evaluation of Software Architecture: ATAM

Certainly, any software development process will involve some method of evaluating the design documents. A formal method for analyzing the architecture which involves a representative sampling of stakeholders, however, is frequently lacking in industry [CKK02, Pau02].

This section presents an overview of the Software Engineering Institute's *Architecture Tradeoff Analysis Method* or ATAM. ATAM grew out of software engineering research aimed at providing a more structured method to assess the ability of an architecture to determine suitability of an

architecture to meet the requirements of the system for which it is designed. Clement's et. al suggest that an architecture is "suitable" if it meets two criteria:

- The system that results from it will meet its quality goals. That is, the system will run predictably and fast enough to meet its performance requirements. It will be modifiable in planned ways. It will meet its security constraints. It will provide the required behavioral function. "[The] architecture is considered suitable if it provides the blueprint for building a system that achieves those properties [CKK02]".

- The system can be build using the resources at hand: the staff, the budget, the legacy software (if any), and the time allotted before delivery. That is, the architecture is feasible from an implementation perspective [CKK02, BM97].

The ATAM Steps

Detailed explanations of the ATAM process can be found through the Software Engineering Institute [Sof06] and in work by Clements et al. [CKK02]. The goals of ATAM are to ensure suitability of the architecture by incorporating input from stakeholders in an organized and disciplined manner [Sof06]. The nine steps in this evaluation process are listed with a brief explanation of each step. The benefits of this method are discussed in Section 5.2 with regard to providing an intuition of how architecture can be used to estimate risk. This method builds on the experience of software engineering research in architecture, and on other methods for evaluating software architecture, notably SAAM (Software Architectural Analysis Method), another scenario based approach [CKK02].

1. **Present the ATAM** Stakeholders are given an overview of the ATAM process for the purpose of explaining individual roles, setting expectations and answering questions.

2. **Present the business drivers** A review of the business goals motivating the design of the system, its most important functions, constraints and quality attributes which are given significant weight.

3. **Present the architecture** Normally conducted by the lead architect, this presentation informs stakeholders of technical constraints, the system context, the major components and their relationships. The focus of this presentation is how the architecture addresses the business drivers.

4. **Identify the architectural approaches** The architect reviews which styles or patterns (see Section 2.4 are used to address the highest priority quality attributes.

5. **Generate the quality attribute utility tree** This step involves, through scenario analysis, to determine two dimensions related to each quality attribute: the importance of the scenario to the success of the project and the degree of difficulty in successfully achieving the desired scenario results. The ATAM descriptions proscribe a graphical representation of these dimensions in the form of a tree. The SEI researchers suggest using a 0 to 10 scale, or, more commonly, a nominal "low - medium - high" scale to estimate these two dimensions. The result of this utility tree generation is a prioritization of quality attributes so that stakeholders may make informed tradeoff decisions.

6. **Analyze architectural approaches** This builds on step 4, by mapping the approaches used in the architecture to the scenarios used in the generation of the quality attribute tree. At this point, the architect identifies tradeoffs exercised between quality attributes and how those tradeoffs are reflected in the approaches used. At this point, risks to the project can be identified and categorized (normally by the quality requirements they are associated with).

7. **Brainstorm and prioritize scenarios** The purpose of this step is to facilitate discussion among stakeholders concerning the prioritization of scenarios explored in the previous steps. More scenarios are generated and placed into the utility tree, reflecting the iterative nature of the software life-cycle (see Section 2.3 for further discussion of this issue). Prioritization of scenarios is recommended to be accomplished through group consensus, often by voting.

8. **Analyze architectural approaches** Repeating the activities of step 6, the architect explains to stakeholders the decisions made regarding how the approaches used reflect the prioritization of scenarios. In a perfect world, this would be an exact repetition of previous work. Expressing the iterative nature of software development, this step is likely to include insight on how other approaches may be incorporated to more accurately reflect the stakeholder priorities. Significant discrepancies likely lead to the entire ATAM process being repeated.

9. **Present results** A summary of the ATAM process is presented to stakeholders. This includes a review of architectural approaches, scenarios and their prioritization, the utility tree, and project risks.

Assessing the ability of an architecture to deliver on a specific quality requirement begins with constructing *scenarios* or *profiles* under which the system is likely to be subjected. A scenario is a "short statement describing an interaction of one of the stakeholders with the system" [CKK02]. Scenarios are described in terms of stimuli and responses. That is, the set of circumstances which may arise and the desired reaction of the system to those circumstances. The most common type of scenario is closely related to that of use-cases, which are being used to describe requirements with increasing frequency. Scenarios, as used for architecture assessment, attempt to describe not only intended interactions of the specified system, but also attempt to describe situations the system may need to address in the future (ie. "growth scenarios"), as in the case of maintainability and portability requirements. The Software Engineering Institute's research also considers a class of scenarios termed "exploratory scenarios" which describe changes likely to stress the system.

3.3.2 Schedules and Software Development Plan

A key motivation for Paulish's architecture-centric software project planning is that by developing the architecture first, the project manager is able to make much more informed decisions with regard to project schedules and personal schedules. Royce supports this idea by identifying a stable architecture a significant *milestone* in the life-cycle. By providing a basis on which stakeholders can refer to system components and map those to requirements, management is given significant added knowledge when planning the remaining software development. Having identified the major components of the system, the project manager is able to select the correct team necessary to bring appropriate levels of expertise to the project. Due to system components being identified, with interfaces and constraints well documented, other software engineers are able to generate personal schedules with a much higher degree of precision [Pau02].

Paulish suggests that early estimation work take place concurrently with the design of the architecture. Once a suitable architecture is developed and assessed, the input to produce higher fidelity estimation models is in place (see Section 4.2). The estimation models provide management with the data necessary to develop detailed work-breakdown structures, or software development plans. Paulish recommends allowing individual developers, with cooperation from team leaders, to generate personal schedules for areas of responsibility. Because major components and interfaces have been defined prior to this, developers are generally able to produce personal schedules with a claimed accuracy within 20%.

3.4 Transition to Low Level Design

The Carnegie Mellon research showed that the level of detail incorporated into the architecture varied from projects to project, a point identified earlier as being related to the ambiguity in the definition of architecture in Chapter 2. Their research identified the difficulty that software architects had in determining the point at which to stop adding detail. They describe the importance of incorporating a "detail knob" in the architecture design plan as a mechanism of responding to the needs of various stakeholders. Paulish takes an approach of bounding the time given for the architecture development. That is, instead of specifically managing the level of detail present in a project, short deadlines for deliverables are given [Pau02]. They present preliminary results mapping project characteristics to level of detail [FBD06]. Their work with these four projects suggests that the level of detail in the architecture is to progress until sufficient to answer stakeholders questions.

Chapter 4

Estimation and Risk

"Cost, schedule and quality are highly correlated factors in software development. They basically form three sides of the same triangle" [BAC00b]. This insight by Barry Boehm suggests what architecture practitioners preach: considering quality requirements early in the process of development has a profound impact on cost and schedule. The following section provides an introduction to the practice of estimating software costs, usually as a function of the schedule. The practice of software estimation maintains the goal of estimating the size of the project, as well as the *effort* or length of time to project completion. The output of these estimation models is a primary consideration for risk estimation techniques. That is, two large risks being measured are completing the project on time (or at all) and the risk of not finishing on budget. Through the development of architecture, one factor towards reducing these risks is by providing the ability to estimate individual risks with much higher precision. Completing the three sides of Boehm's triangle, this thesis shows how the architecture development process can be used to ultimately lower costs and keep projects on schedule in Chapter 5. Section 4.3.2 presents the risk evaluation technique known as decision trees as one possibility to aid the project manager in choosing alternatives.

4.1 Estimating Size

Size of the software system is the dominant factor used to estimate effort or cost of developing the system. Once the architecture is developed, one can estimate the size of the software which becomes input into effort estimates. Following is an overview of the most common units for expressing

the estimated size of software projects. Almost all size estimates are based on inferences made by project stakeholders drawing from experience with previous projects within the same domain. Software development organizations keep detailed records on project size for comparison purposes. These comparisons are matched as closely as possible by project managers and lead developers to the project being estimated. Ultimately, however, these size estimates involve some degree of subjectivity, as those stakeholders are expected to draw a comparison between the current project and past projects. A desire to remove some of this subjectivity prevalent in "lines of code" estimates has led to measurements in function points, and recently, use cases and object points, which can be inferred more directly from the architecture documentation.

4.1.1 Units of Software Size

This section presents a review of units commonly used to count and estimate the size of the software project. Each of these estimates of size carries a degree of uncertainty. As these size estimates are used as input for estimation models, this uncertainty becomes risk of project completion against schedule and budget. The following sections are to review the reader concerning background material on inputs for estimation models. For readers experienced with this material, the discussion of estimation techniques is resumed in in Section 4.2.

Source Lines of Code

There are two primary units used for estimation of software size: function points and source lines of code (SLOC). Source lines of code have been in practice the longest for use in formal estimation models. However, their variability between languages and individual programmers makes this a somewhat unreliable indicator of actual project size. Research was applied to correlating estimated lines of code to efforts by life-cycle phase. Estimates involving lines of code suffer from the inability to easily account for project development activities other than programming. Defect removal, documentation, communication, project management and change management all fare poorly when estimated using SLOC [Jon98]. The coding phase of the software development life-cycle is generally estimated below 35% of the total effort needed, highlighting the impact of the shortcomings associated with SLOC based estimates [Jon98].

The main reason that SLOC estimates fare so poorly is due to a lack of correlation between lines of code and complexity of the project. Today, high level languages are capable of producing

extremely complex logic with highly terse code. Further, significant variation between programming styles can greatly influence the number of lines of code produced by a given programmer, even when policies attempt to enforce uniform style constraints. A skilled developer, proficient in the language and development environment will often be able to produce the same functionality with fewer lines of code as a novice programmer under identical circumstances. Further, the advent of graphical programming tools allow complex logic to be generated with very few lines of code augmenting a drag and drop interface.

Some issues regarding estimates based on lines of code have been addressed through refining the definition for *line of code*, and subsequently categorizing the code counts into classes. The goal of this effort is to improve communication and repeatability. That is, metrics built on a given SLOC definition should be aware of exactly what types of statements are counted, and the SLOC should be the same if the count is repeated. Blank lines and comments are generally excluded from these counts, however different techniques differ in their approach to treating compound statements, function headers and prototypes, lines containing only a delimiter, modified and code not programmed by the development team (eg. library code, generated or converted code, code in other languages) [Par92]. Lines of code remain a source of uncertainty for use in estimation models due to differences in languages, coding styles and the amount of project effort required other than coding.

Function Points

Function points were introduced by A.J. Albrecht at IBM as a way of expressing the size and complexity of software systems independent of implementation language, technology and capability of the project team. Function points were originally intended as a device to allow for precise effort estimation, but they remain a unit for discussing the size of a software system. Function points generally consider five areas, which can be derived as early as the end of the requirements stage:

1. system inputs
2. system outputs
3. interactive inquiries
4. external logical files
5. internal logical files

Function point analysis involves counting the number of each attribute instance and applying a weighting scheme. Most practitioners consider these attributes to have varying levels of complexity, and to reflect this lack of consensus, a number of schemes have been proposed and practiced [Jon98]. Once a function point total is calculated, some multiplier is applied to the total as a reflection of the perceived intricacy of the overall system. This multiplier considers factors such as the scope, class and type. These complexity adjustment factors tend to be highly subjective, and much research has been done in the area of software complexity metrics, which attempts to finds methods of quantifying complexity [Zus97, Jon98].

Albrecht's original system has been adapted and modified several times since its inception in the 1970's. The International Function Point User's Groups was founded in the early 1980's to further explore this method of sizing software projects and develop a set of standards for the practice [Int07a]. Other methods include the Boeing 3D function point, DeMarco bang and Mark II. These techniques differ from the original systems by varying the amounts of weighing for different complexity estimates [Jon95]. Many of these methods have attempted to map their function point method against lines of code by programming language. These indirect methods of obtaining SLOC estimates are widely employed in estimation models, due to popularity of the SLOC unit among software managers [Roy98, Jon95]. Function points suffer from inaccuracies as measurements of size due to the inequality of the types of attributes, and inaccuracies of those attributes mapping back to project size.

Use Cases

In recent years research has been conducted in estimating software size by use cases. These methods benefit from the widespread use of the Unified Modeling Language (UML) and object oriented design. Use cases, as a method of describing the requirements, offer a bridge between requirements and design [Mac01]. Much of the research in this area has looked at automating sizing estimates by using models generated through currently established processes such as the Rational Unified Process [Kru02]. This practice of applying use cases as a manner of estimating size is still in early stages of adoption; however, preliminary research suggests that use cases can be effective in estimating size [ADSJ01].

Research conducted with use cases involved three industrial studies by Anda et al. at an e-commerce company [ADSJ01]. Their approach involved assigning weightings to the actors and

use-cases in use case models derived using the Rational Unified Process. Approaches involving use-cases closely model those employing function points, that is, weightings are given to the countable features (categories of use-cases, actors) and weightings are then applied based on some criteria. A final adjustment is then made to the use case point total as an attempt to reflect the overall complexity of the system, development environment, as well as maturity and experience of the staff and organization. Approaches to these weightings have taken place based on complexity metrics [CS], number of transactions [MAC05a], types of interfaces [ADSJ01] and heuristics applied by the development team [MAC05a].

Modern applications of agile development, more integrated object-oriented processes and web-based systems have lately given rise to more units for measuring size. Object points, web-objects, application points and multimedia points all attempt to address modern practices [Rei00a]. Few of these have been used as part of formal estimation models on large scale projects [Rei00b]. However, recent trends have led to smaller systems being developed, and potential exists for these other units to be useful in many domains [Jon98].

4.2 Estimating Effort and Cost

While estimating the size of a project becomes the first step towards estimating cost, other formal estimation methods exist to bound cost. Effort is generally measured in *work-months* (programmer-months, developer-months, man-months), or just *months*. Cost is then calculated by multiplying effort by burdened rate (ie. salary, overhead) of the staff employed to carry out the task. These costs are calculated by phase in the software life-cycle. These estimates can then be used as input for project management (eg. scheduling, work breakdown structure) and business processes (eg. forecasting, marketing) [Roy98]. Most software cost estimation techniques, and the associated databases of prior projects used to calibrate them, are proprietary tools of the companies who use and develop them. Software engineers have published information on the tools used to estimate software cost, as well as samples of the data and techniques used to calibrate the weights [Jon95].

The following sections present methods for estimating effort in software projects which are used to estimate cost, allocate resources and determine the project schedule. The reader familiar with effort estimation is invited to proceed to Chapter 5 which discusses the impact of architecture-centric project management on reducing the uncertainty present in the estimation process.

4.2.1 Estimation By Analogy

Manual methods are generally based on heuristics, or sometimes termed, "rules of thumb." The project knowledge accumulated over time is used to predict the outcome of future projects. The ease of producing a manual estimate remains the most popular choice for project managers, despite evidence to their inaccuracy [Jon98]. The influence of optimism on this process was noted as a significant cause of overrun in Chapter 1, and can be a strong influence in manual estimation techniques. Most of these rules attempt to provide an average amount of coding per programmer-month based on SLOC. Many software development groups maintain extensive databases, recording project management data so these rules may be refined over time [Roy98]. This allows for tracking of performance of varying teams, combinations of teams, and incorporates the ability to track the SLOC per work-month having fixed a given language, framework or other factor. Careful record keeping also allows for tracking of work-months over phases in the life-cycle. This activity based analysis allows for domains with highly different processes to generate tailored estimates based on a history of the activities associated with that domain.

Manual methods which employ function points frequently use templates for rating the various categories of function points. These categories are given a weighting based on their *scope*, *class* and *type*. Table 4.1 shows a sample taxonomy of these properties that are used by Software Productivity Research [Jon98]. The weight is assigned to a function point based on the sum of its scope, class and type weights. The sum of this function point total is computed, and then scaled with adjustment factors determined by domain specific or team specific properties.

The weighted function point total is then usually applied to a series of charts mapping function points to SLOC by language, pages of documentation, testing cases, predicted number of bugs and ultimately, development schedules. Rather than mapping function points directly to programmer-months, however, function points are frequently expressed in a mapping to calendar months. A separate mapping relates the number of personnel to the number of function points. Thus, function point methods attempt to address issues related by Fred Brooks involving the perils of adding developers late in the process. By attempting to correctly map not the length of development time in calendar months independently of the number of developers, this method is often felt to be a greater tool in scheduling.

Weight	Scope	Class	Type
1	Subroutine	Individual software	Nonprocedural
2	Module	Shareware	Web applet
3	Reusable module	Academic software	Batch
4	Disposable prototype	Single location-internal	Interactive
5	Evolutionary prototype	Multilocation-internal	Interactive GUI
6	Standalone program	Contract project-civilian	Batch database
7	Component of system	Time sharing	Interactive database
8	Release of system	Military service	Client/server
9	New system	Internet	Mathematical
10	Compund system	Leased software	Systems
11		Bundled software	Communications
12		Marketed commercially	Process control
13		Outsource contract	Trusted system
14		Government contract	Embedded
15		Military contract	Image processing
16			Multimedia
17			Robotics
18			Artificial intelligence
19			Neural net
20			Hybrid: mixed

Table 4.1: SPR's multipart taxonomy of function points

4.2.2 Parametric Models and Methods

Parametric models attempt to account for a number of factors (parameters) which influence the effort estimation. The ability of a parametric model to reflect the impact of these factors allows us to understand how the process of architecting can influence the effort estimation (Section 5.1). The wide acceptance of the COCOMO II model for estimation in both industry and academia make it suitable for consideration as an input for risk evaluation. The openness of its algorithm has allowed significant research with COCOMO, results that are unavailable with other, proprietary techniques. Capers Jones remarked that "COCOMO remains the only software estimation model whose algorithms are not treated like trade secrets" [Jon95]. There exist a number of vendors providing premium estimation services and software based on the COCOMO model. Among these are COSTAR by Softstar Systems (`http://www.softstarsystems.com/`), Cost Xpert by Cost Xpert Group (`http://www.costxpert.com/`) and ESTIMATE Professional by the Software Productivity Center (`http://www.spc.ca/`).

COCOMO II

The most frequently referenced automated method for large scale software estimation is the COCOMO (COnstructive COst MOdel) II strategy [Roy98, Fai06]. This strategy, built on the original COCOMO method by Boehm [BAC00a], incorporates changes reflecting modern software engineering practices. The model, like those built by analogy, requires a calibration of the application domain and environment. This calibration is based on many of the same factors that go into determining weightings for function points in sizing techniques. After calibration with over 161 project data points, the COCOMO II model gave predictions of effort within 30% of the actual results 75% of the time [BAC00a]. This approach allows for statistical interpretation of historical data intergraded with expert judgement.

The COCOMO II strategy defines three models for cost estimation: The *prototyping model*, *early design model* and *post-architecture model*. These models corresponds to increasingly detailed levels of input through the phases of the software life-cycle. Each of these assumes successive levels of information exists, brought about through the reaching of decisions. This added knowledge reflects the risks at that point in the project.

The prototyping or "applications composition" model is designed to provide an "order of magnitude" approximation during the feasibility assessment phase. The early design model is designed

to provide estimates concurrently with the formation of the architecture and formalization of the requirements. This model uses the same base equation to estimate effort as the highest fidelity, post-architecture model, and the point at which the models differ is at the precision of the individual variable estimates. The post-architecture models assumes that the project currently has a "stable" set of requirements[1] and a candidate architecture. Just as with proprietary estimation models, most organizations employing models such as COCOMO II still do not make public their calibration data [Jon95, BAC00a]. The numbers discussed in the following section reflect typical values cited in academic papers. The notion of architecture adopted in this thesis generally involves a level of detail approximate to that of the COCOMO II post-architectural model.

The COCOMO II post-architectural model uses the base estimating equation,

$$\text{Effort} = 2.45 \prod_{i=1}^{17} (EM_i)\,(Size)^{1.01+\sum_{j=1}^{5} SF_j}$$

where

$$\begin{aligned} \text{Effort} &= \text{number of staff-months} \\ EM_i &= \text{refers to process scale factors, shown in Table 4.2} \\ Size &= \text{number of KSLOC, for more see footnote}^2 \\ SF_j &= \text{refers to effort scale factors, shown in Table 4.4} \end{aligned}$$

Scale Factor Symbol	Abbreviation	Name
SF_1	PREC	Precendentedness
SF_2	FLEX	Development Flexibility
SF_3	RESL	Architecture and Risk Resolution
SF_4	TEAM	Team Cohesion
SF_5	PMAT	Process Maturity

Table 4.2: COCOMO II process exponent scale factors [BCH+95]

[1] the term "stable" when applied to requirements is used to indicate that a formal requirements engineering phase has occurred, not that the requirements are *frozen*. In fact, Boehm indicates that requirements creep is a built in factor in all COCOMO models [BAC00a].

The exponent by which COCOMO scales the software size is determined by the combined effects of five "scale factors" or cost drivers, shown in Table 4.2. These five factors reflect concerns such as the degree of domain experience of the organization and the degree of flexibility in the process, company, contract and communications. Table 4.3 shows the factors Boehm uses to assign values to the five exponent scale factors. Several of these factors are qualitative in nature, and very dependant upon the opinions of the members providing input for the estimation. The coefficients listed in Table 4.4 are assigned values between 0.5 and 1.5. These effort adjustment parameters are calibrated by maintaining a database of those controls over a series of projects. Ongoing research in the area of software estimation involves improving on these estimates. A large amount of this research involves using metrics to estimate product complexity [BAC00a, Jon95].

Use Case Based COCOMO

Mohagheghi et al. [MAC05b] conducted an industry experiment using *Use Case Points* which generates an estimate of effort in worker-months based on use cases which specify requirements. There approach was to implement a complexity assessment of actors and use cases based on technical and environmental factors, similar to function point analysis. A weighted use case point total is input into a formula derived from COCOMO II's formula for adapted software. This specific use case method was tested on six moderately sized projects. Similar methods were developed by Anda et al. [ADSJ01], and Carroll of Agilis Solutions [Car05], who claims effort estimates within 9% of 95% of over two hundred large projects. Estimation based on use cases is a fertile area of research as the popularity of object oriented design and UML continues to grow, this approach attempts to take advantage of a design artifact already in popular use. The ability of use-cases to be used as part of the architecting process in particular adds relevance to claims that at this point in the life-cycle a significant amount of knowledge is gained which enables risk reduction.

Other Parametric Effort Estimation Models

There exist numerous other parametric models which attempt to address qualities not covered by COCOMO II, notably, those related to domain specific attributes. Included among these are SLIM, Checkpoint, Price-S, ESTIMACS, SEER-SIM and SELECT. SELECT stands alone among these with explicit consideration of security issues. All are calibrated from analysis of past projects, hence, like COCOMO II fare poorly with unprecedented software projects. As noted previously,

Symbol	5	4	3	2	1	0
SF_1	thoroughly unprecedented	largely unprecedented	somewhat unprecedented	generally familiar	largely familiar	thoroughly familiar
SF_2	rigorous	occasional relaxation	some relaxation	general conformity	some conformity	general goals
SF_3	little(20%)	some(40%)	often(60%)	generally(60%)	mostly(80%)	full(100%)
SF_4	very difficult interactions	somewhat difficult interactions	basically cooperative interactions	largely cooperative	highly cooperative	seamless interactions
SF_5	Based on CMMI Rating					

Table 4.3: COCOMO II process exponent values and criteria

Scale Factor Symbol	Abbreviation	Name
EM_1	RELY	Required Reliability
EM_2	DATA	Data Base Size
EM_3	DPLX	Product Complexity
EM_4	RUSE	Required Reusability
EM_5	DOCU	Documentation Match to Life-cycle Needs
EM_6	TIME	Time Constraint
EM_7	STOR	Storage Constraint
EM_8	PVOL	Platform Volatility
EM_9	ACAP	Analyst Capability
EM_{10}	PCAP	Programmer Capability
EM_{11}	AEXP	Applications Experience
EM_{12}	PEXP	Platform Experience
EM_{13}	LTEX	Languages and Tool Experience
EM_{14}	PCON	Personnel Continuity
EM_{15}	TOOL	Use of Software Tools
EM_{16}	SITE	Multi-Site Development
EM_{17}	SCED	Required Development Schedule

Table 4.4: COCOMO II effort adjustment scale cost drivers [BCH+95]

most of these are proprietary models, not readily available for academic research [Jon98, BAC00a].

Effort Estimation and Architecture

The previous sections examined common sizing and effort estimation practices with focus on one parametric model, COCOMO II. Effort estimation, particularly with the COCOMO model, has been shown to be more accurate at the point of architecture, supporting the claim that at this point in the software life-cycle managers are able to reduce a substantial amount of risk through gained knowledge of the system. The following section introduces concepts associated with risk analysis, which allow for formal analysis of specific identified concerns of a project. Chapter 5 examines specific risks addressed through the architecting of the system and how those are reflected in the COCOMO model.

4.3 Risk Analysis

Risk analysis is the practice of identifying and estimating the risks associated with a given activity. The following section provides the reader an overview of risk analysis practices, can be used in conjunction with architecture-centric project management. The risks identified during the architecting of the system, specifically through architectural evaluation such as ATAM (see 3.3.1), can be used as input for the types of analysis introduced in this section. ATAM provides stakeholders a process to identify system risks, particulary those associated with quality requirements, and the impact of those risks on the system as a whole. Most information in this section concerning risk analysis is drawn from Robert Charette's work [Cha89] and the reader is referred to that text for a more in-depth treatment of the subject. The reader experienced with risk analysis is invited to proceed to Chapter 5 which examines the specific risks addressed through architecture-centric management.

Charette suggests that each risk can be identified by an ordered triplet of information:

1. s_i, the the risk as categorized by type
2. l_i, the likelihood, or probability, of the risk occurring
3. x_i, the severity, or consequence of the risk [Cha89]

To that end, one can categorize the threats associated with a software project as those factors which may cause the project to not succeed. These may include failure to meet schedule (or,

consequently budget), failure to deliver functionality, and failure to deliver quality requirements (for an overview of specific quality requirements see Section 2.2. Before examining how the practice of architecture-centric project management can alleviate these risks, a discussion follows of how risks are estimated and evaluated. Chapter 5 examines which risks exist to software projects and how they can be reduced and evaluated through this process.

As the criteria suggest, each identified risk has two primary components: the chance of occurrence and the magnitude of the consequences. This view of risk is related to the statistical notion of "expected value," or what the outcome of an alternative will be should it repeated numerous times [Cha89]. The goal of risk analysis is to provide the project manager with a tool to assist in deciding on a course of action. The net result of these analysis techniques is a comparison of the risks associated with various courses of action, usually displayed graphically or numerically. As addressed in Section 1.2, optimism plays a large factor in project overrun with stakeholders generally assuming that unforseen negative events will not occur. Risk analysis provides project managers a tool to numerically evaluate alternatives and remove some degree of optimistic bias from the decision making process.

4.3.1 Expected Values

One method of evaluating the risks associated with different alternatives is to consider the values associated with each outcome and the probability of that outcome occurring. Expected value is the sum of all probabilities of a risk's outcomes occurring multiplied by the value of that outcome's occurrence. For a random variable, X, expected value is expressed mathematically as,

$$E(X) = \int_{\Omega} X dP$$

where Ω is the space of all possible values of X and P is the probability space of X. For discrete random variables, this simplifies to,

$$E(X) = \sum_{i} l_i v_i$$

where i is the number of possible outcomes, l_i is the probability (likelihood) of outcome i occurring and v_i is the value (payoff, consequence) associated with outcome i. Note that the expected value may possibly never equal any possible value of the random variable[3]. Likelihoods represent proba-

[3]The classical example of this is with the role of a six sided die where, $E(\text{die roll}) = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = 3.5$

bilities, and thus the all likelihoods expressed should sum to 1 [MA02]. Expected values are used in insurance, gambling and business extensively in order to provide decision makers with a tool to consider the balance between risk and reward.

In order to consider calculating expected values of alternatives, one must first have estimates for the likelihood and values associated with a given alternative's consequences. The accuracy of those estimates will have a large impact on the expected values. Consider a simple example related to software architecture, treating estimation error as a discrete random variable. Assume the architecture team for our company, Large Software Project R Us (LSPRU), proposes a given design which is termed architecture *A1* and following the application of the COCOMO II post-architecture analysis the effort is then calculated in monetary terms. It is assumed that there exists a database containing a reasonable set of relevant project outcomes and estimates, with accompanying regression analysis for use in calibration. Based on previous use of COCOMO with similar projects at LSPRU, one can estimate the possible outcomes of this project with respect to finishing on schedule, as well as the likelihood and value of those outcomes. These are shown in Table 4.5.

Outcome	Likelihood	Cost
1 month early	0.05	$-4,500,000$
on time	0.25	$-5,000,000$
1 month late	0.2	$-5,500,000$
2 months late	0.15	$-6,000,000$
3 months late	0.1	$-6,500,000$
4 months late	0.05	$-7,500,000$
5 months late	0.05	$-8,000,000$
6 months late	0.05	$-8,500,000$
7 months late	0.05	$-9,000,000$
8 months late	0.05	$-9,500,000$

Table 4.5: Estimated outcomes of project at LSPRU

Applying the definition for expected value of a discrete random variable, where l is the likelihood of occurrence and v is the value or consequence of the loss (expressing cost in millions of dollars):

$$E(X) = \sum_i l_i v_i$$

$$
\begin{aligned}
E(X) &= (0.05)(-4.5) + (0.25)(-5) + (0.2)(-5.5) + (0.15)(-6) + (0.1)(-6.5) + (0.05)(-7) \\
&\quad +(0.05)(-7.5) + (0.05)(-8) + (0.05)(-8.5) + (0.05)(-9) \\
E(X) &= -6.125
\end{aligned}
$$

Given these calculations, the project manager estimates the cost of implementing this architecture at $6,125,000. Realistically, the random variable represented time to complete the project would be treated as a continuous random variable, and the cost associated with project overrun may not increase linearly with time. In an ideal world, the project manager could compare this architecture with other alternatives, examine the expected values of each, and then make a decision with sound numerical backing.

4.3.2 Decision Trees

This section presents decision trees as one method for analyzing risks associated with various alternatives. Decision trees reflect well the choices of alternatives associated with decisions made during architecting. As each decision is made, knowledge is gained about the project, which allows for more closely identifying the likelihood and outcome of consequences associated with these alternatives. Further, trees in general provide a familiar representation to computer scientists. Other mechanisms exist for risk analysis, and are mentioned in 4.3.2.

Decision trees provide a graphical means of comparing alternative courses of action with their probably outcomes and expected values. This method of risk evaluation provides the benefits of being simple, as it is closely related to the notion of expected values. Root and internal nodes of the decision tree represent alternatives, while edges (other than those extending from the root node) represent probabilities. Note that the sum of the probability edges coming out of a given node must sum to one (100%). The leaf nodes of the decision tree represent the payoff amount of a given decision. The advantage to using decision trees for risk assessment is that stakeholders can easily visualize all alternative courses of action and the consequences of each. In the case of multi-level trees, the probability of a given outcome is computed by taking the produce of all probabilities along the edges starting at the root and continuing to the outcome[4].

Revisiting the example from Large Software Projects R Us, assume the project manager gets her wish, and is presented with three different architectures. Each of these has been estimated using

[4] This is termed *conditional probability*. For an in-depth treatment of this subject, the reader is referred to [MA02]

Alternative	E(X) Calculations	Expected Value
A1	$(0.3)(-4.75) + (0.7)(-7.5)$	$-\$6.675M$
A2	$(0.1)(-4) + (0.9)(-7)$	$-\$6.7M$
A3	$(0.2)(-5.5) + (0.8)(-6.5)$	$-\$6.3M$

Table 4.6: Expected Values of LSPRU Alternatives

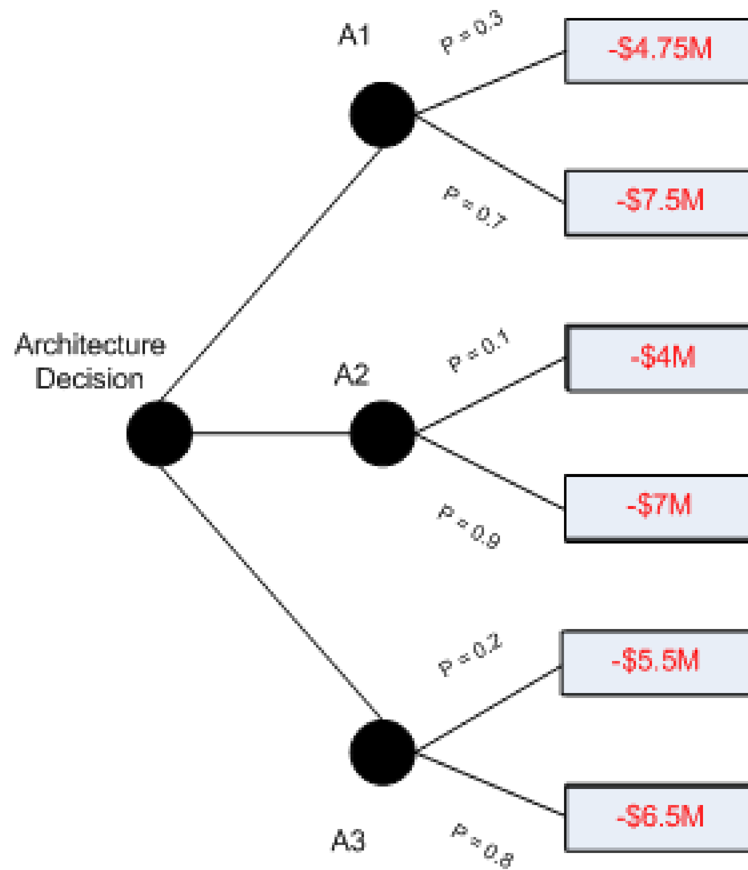

Figure 4.1: Decision Tree for LSPRU Architecture

their favorite cost estimation model, and the results have been aggregated into a decision tree. The tree is simplified by aggregating each outcome as either **early/on-time** or **late**. Due to the nature of each alternative architecture, however, the probabilities of finishing the project early/on-time (represented as the uppermost outcome associated with each node) and probabilities of finishing late (lower outcome) vary significantly. This tree can be seen in Figure 4.1, and each alternative presents itself a highly different expected value. The first architecture, A1, has a 0.3 probability of finishing on time, with a cost of $\$4.75M$ while the cost to develop the second architecture on time is only $\$4M$, however, the estimated probability of this falls to 0.1. Expected values now provide a way to compare these alternatives. The comparison is presented in Table 4.6. From this table, note that architecture A3 seems to have a lower expected value than the others.

Making the example somewhat more realistic, lets assume that the first architecture, A1 comes with a few different alternatives. Perhaps the architects are split on choosing a framework. Some developers feel certain that by implementing in J2EE that the probability of completing the project on time increases significantly due to their experience with it. Others have concerns that in order to compensate for inherent performance issues in the J2EE framework, that work-arounds will take longer to implement, and choose instead to use .NET. This would lead us to produce a multi-tiered decision tree as described earlier, adding to the complexity not only of the graph, but of the decisions necessary to resolve by stakeholders. Further, these multi-tiered decision trees can be used to visualize the potential impact of cascading decisions[Cha89].

Other Methods for Risk Evaluation

Decision trees provide a simple method for visually analyzing the alternatives available and associating their respective probabilities and outcomes. However, as noted, decision trees may become arbitrarily large and complex, and fail to account for multiple dimensions of risk. Charette suggests *isorisk countour maps* as an alternative. These maps graph curves against axes representing costs and probability. They enable the project manager to view several alternatives on the same graph simultaneously; however, this requires the establishment of a risk referent, or risk tolerance, level. Project management processes, such as those based on network models (eg. PERT, Petri-nets, queueing models), also incorporate a degree of risk evaluation. Another software engineering tool for project management is the *work breakdown structure* (WBS), a detailed description of all significant work, task decomposition, assignment of responsibilities and framework for scheduling and budget-

ing [Roy98, Boe84]. The process of creating a well planned WBS is considered by practitioners to be a large step toward risk evaluation [Cha89].

4.4 Conclusion

The presence of multiple alternatives within an architecture presents one very large factor which increases the complexity. Other realistic issues in risk evaluation include estimates of the probabilities involved. Large corporations keep detailed databases of effort histories with regard to software projects, and these used in conjunction with estimation models provide estimates of those probabilities, although it should be noted that relatively small changes in probabilities can have a large impact on expected values. Techniques such as the Delphi method have been developed to refine expert input from multiple sources in order to produce accurate estimates of these probabilities [KCLS98] [Cha89]. No method is perfect, and the large complexity and exponential number of alternatives associated with software projects provides limits on how tight of estimates any method may ever obtain [Bro95, Cha89]. Risk assessment does not offer any “silver bullet” [BAC00b] to project managers in order to facilitate a successful project; however, evidence has shown that by careful consideration of risk factors and sound software engineering techniques, higher success rates can be achieved than results show [AV02, Cha89].

Chapter 5

Addressing Risk with Software Architecture

The premise of this research is that at the point of architecture, a significant amount of risk is removed from the project in relation to the effort expended to this point. Naively, it can be assumed that all risks associated with a software project naturally decrease as the project moves through the software life-cycle. This decrease in risk is a direct function of the amount of knowledge gained. However, savvy project managers are aware that this increase in knowledge and hence, reduction in risk, does not occur uniformly by life-cycle phase. The success associated with software architecture, notably that reported by Daniel Paulish and others at Siemens Corporation [Pau02, NPSS01], as well as academic research by Jan Bosch [Bos00], and the work of those at the Software Engineering Institute [BCK98, CKK02], suggests that at the point of architecture, a significant decrease in the amount of risk associated with a project. This decrease could be visualized as an "inflection point" on the curve of risk measured as a function of time. The exact point at which this inflection may occur remains unknown, for a number of reasons:

- Formal risk assessment is still not a widespread practice in industry
- Organizations engaging in formalized estimation and risk assessment methods tend to keep this data proprietary.
- Current practices of estimation and risk assessment generally fail to account for quality re-

quirements

- The extent to which an architecture is detailed varies significantly based on domain, project size, scope and stakeholder preference. This point is elaborated in Section 2.5.3.

- The nascent nature of software architecture is only beginning to show results in software industry.

Given that as knowledge of the project increases over time, and with this knowledge comes a reduction in the level of uncertainty inherent in the process, this added knowledge provides the project manager with higher precision estimates. At early stages, when little is known about the project, estimates are highly inaccurate and large potential exists for unforseen problems to occur and for quality attributes to be unaccounted for. Early decisions made in the software life-cycle remove uncertainty from the project, hence lower risk by identifying and constraining. However, poor decisions at an early stage have been shown to be extremely expensive, with this expense growing the later in the life-cycle that the mistake is discovered. Conversely, early quality decisions enable the team to avoid this expense. Through a solid architecting process, including a formalized evaluation involving most stakeholders, the project team can therefore remove significant risk from the project at an early stage. The premise suggests this provides a large amount of risk reduction for the effort expended. To this end, this chapter presents an accumulation of how currently available work suggests risks can be largely reduced or bounded through the process of architecture-centric project management.

5.1 Addressing Estimation Risks

One of the largest and most costly risks associated with large software projects is that of not finishing on time or, consequently, on budget. As detailed in Section 1.1, a high percentage of large projects run significantly over schedule and over budget, costing corporations millions of dollars, as well as damage to reputation. It is well known that the estimation techniques discussed in Chapter 4 carry with them a certain amount of uncertainty [Jon95, BAC00a]. The amount of this uncertainty decreases throughout the life-cycle, and, according to COCOMO pioneer, Barry Boehm, this uncertainty decreases dramatically during the design process as can be seen in Figure 5.1. Graphically, the figure demonstrates the decrease in uncertainty, shown as the divergence between

the curves, as the software life-cycle progresses. Early estimates at the product definition stage are shown to vary between one-quarter the actual effort and four times the actual effort. By the time, design specification is complete, this variance in estimation error ranges between eight-tenths and a fourth-again as much of the final effort.

This observation has been confirmed by several others, including Paulish [Pau02]. COCOMO II addresses this by considering estimation to be an iterative process, to be repeated multiple times as more information is known about the project. That is, as more knowledge is gained in the life-cycle, uncertainty is removed from the process, which provides the manager with better estimates, equating to lower risk. In reality, formal estimation techniques are rarely done at all, and when completed, usually occur only once during the life-cycle [Jon95].

Research has well established that the further along in the software life-cycle, the more costly it is to incorporate changes to the system [BAC00a]. Boehm's research further suggests that as the process moves through the lifecycle, that the cost of these changes increases exponentially, as illustrated in Figure 5.2. Motivated by this, software engineers seek to create an architecture which will not only meet the functional and quality requirements specified, but also allow for future modification with as little expense as possible.

Examining the COCOMO II process scale factors (Table 4.2) and effort adjustment factors (Table 4.4), it becomes apparent that a number of these factors can be estimated more closely following the steps outlined by Paulish's *architecture centric project planning approach*. The level of *precedentedness*, SF_1, should follow immediately from the architecture deliverables. Precedentedness is defined as "the degree of domain experience of the development organization" [Roy98].

Capturing the degree to which the architecture-centric process has been implemented by the team is SF_3, *architecture and risk resolution*, whereby following the prescribed techniques of ATAM and subsequent risk analysis as described in Section 4.3, an economical adjustment factor can safely be applied [Roy98]. The impact of these two factors is large, as COCOMO incorporates these into the exponential scaling factor, not simply the multiplicative.

Among the cost drivers whose boundaries become more apparent are EM_2, *data base size*. This factor is calculated by a ratio of data base size (in bytes) to software size (in SLOC) and is bounded through the development of architecture by determining the software components and database usage prior to cost estimation. EM_3, *product complexity* can be estimated by applying established software metrics to the architecture. This degree to which this particular cost driver can be estimated

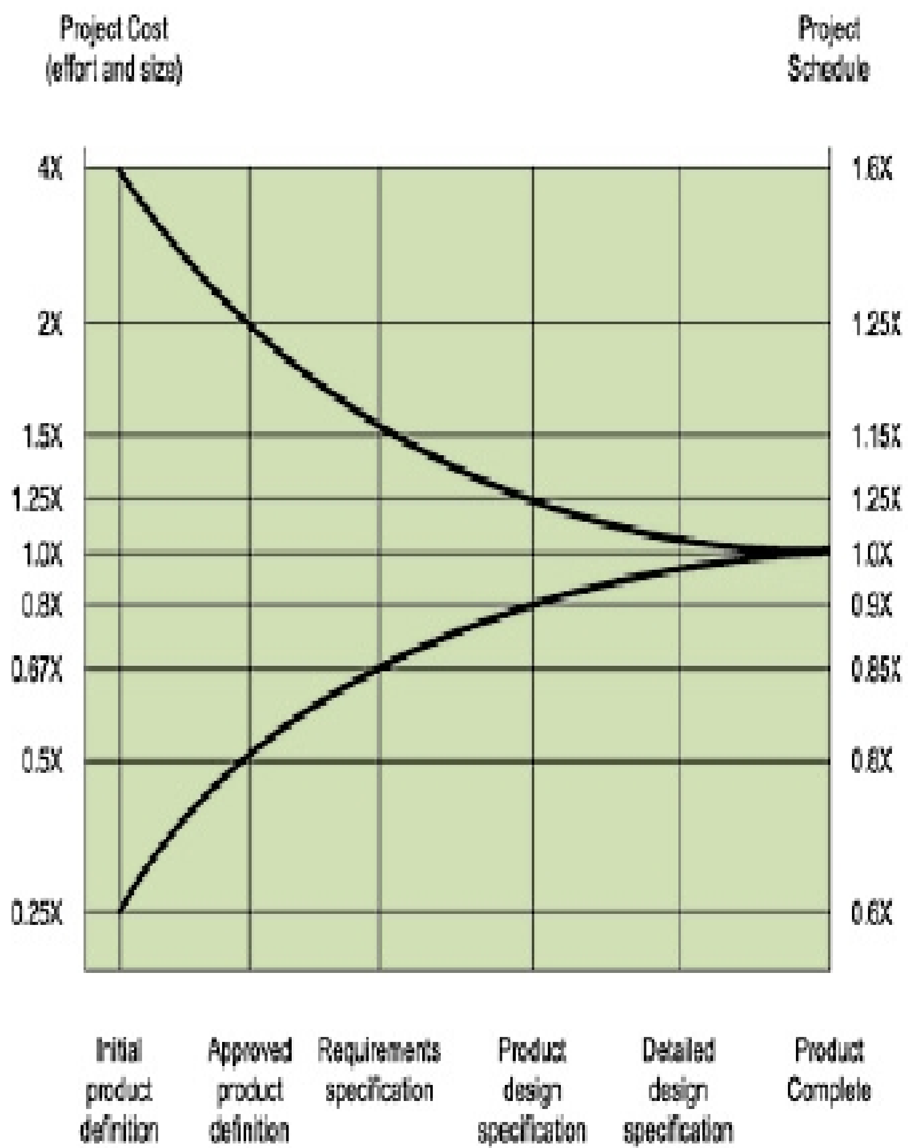

Figure 5.1: Error Associated with Software Estimation [MIT02]

is highly dependant on the level of specificity in the architecture. Research of Fairbanks [FBD06] and Bosch [Bos00] seems to suggest that this indicator will likely be a place where the project manager may be required to fine-tune the "detail knob" for specific components, in order to closely estimate the effort.

EM_8, *platform volatility* refers to the hardware and software components making which system resides on and calls. Components such as the network, operating system, database management system, and framework make up the platform used in this COCOMO cost driver. The volatility of these components is determined by estimating the frequency of major and minor changes during the development life-cycle. Again, once the architecture has been developed, the project manager is in a position to closely estimate this factor based on history.

EM_{11}, *applications experience*, EM_{12}, *platform experience* and EM_{13}, *languages and tool experience* all follow directly from the architecture development process. That is, once decisions have been reached regarding these, the project manager is in a far better position to estimate the level of experience and competency of the team with respect to the applications, platforms and languages. Further, should the desired experience not be present on the proposed team, in some cases the manager may be able to lower this cost driver by bringing on board an individual with expertise in a given area, allowing the cost driver to be lowered at an early stage. By determining this early, the team is able to avoid the temptation of making "a late project later [Bro95]" by onboarding an expert further in the life-cycle.

EM_1, *required reliability*, EM_4, *required reusability*, EM_6, *time constraint*, EM_7, *storage constraint* are often considered to be quality requirements. This effect and importance of these requirements will be determined through the ATAM steps. Specifically, the quality attribute tree generated through ATAM gives an indication of the importance of these constraints to all stakeholders.

In summary, the ability to bound project estimation much more closely following the development of the architecture is widely accepted [Boe84, Pau02, Bos00]. The ability to more closely estimate the time and cost for developing the system can be easily equated to the risk of developing the system on time (or equivalently on budget) [Jon95]. While completing the project, as specified in the requirements, on time and on budget is only one risk, it is considered by most project managers to be the most significant risk [Roy98]. Further, the number of instructions programmed is by far the largest factor identified as an indicator of project effort [BP88b]. By investing the resources into developing the architecture, the project manager reduces the likelihood of developing unnecessary

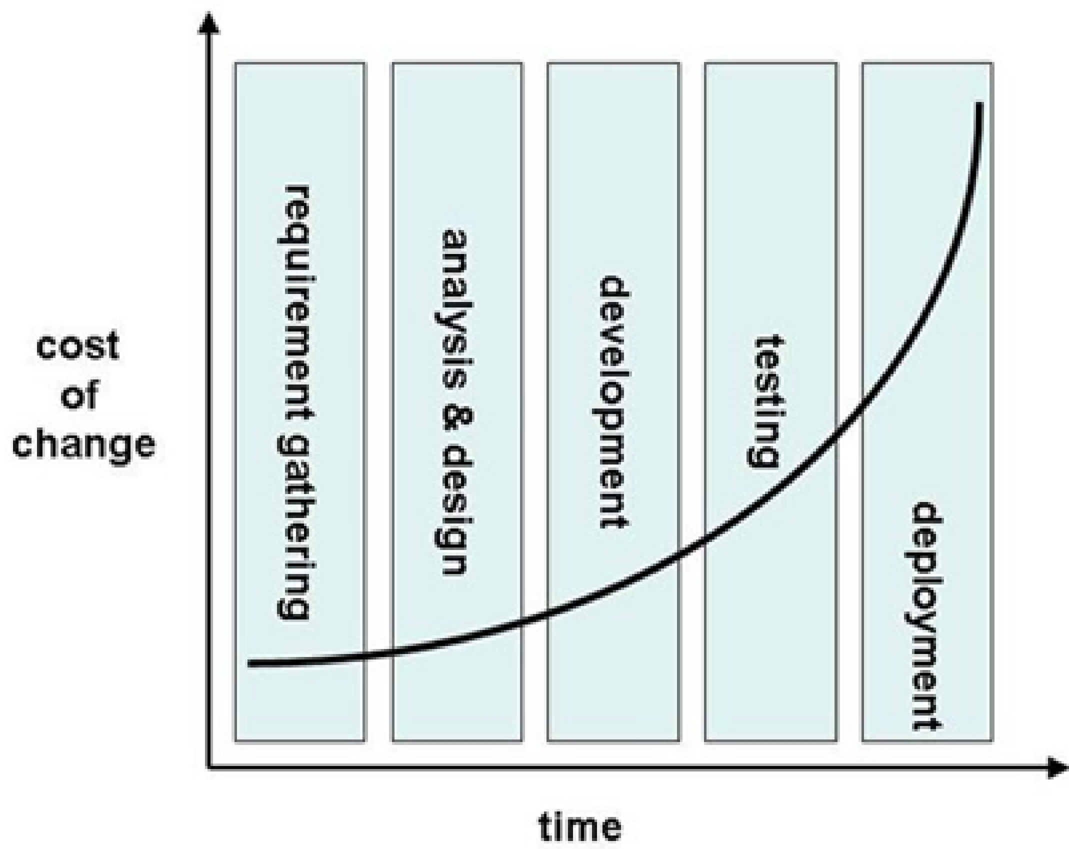

Figure 5.2: Cost of Change Over Life-Cycle [Met05]

components.

5.2 Addressing Quality Risks

The ability of architecture to provide an early indicator of quality requirements is a large motivation for architecture-centric project management. Most published work on the subject of architecture evaluation and analysis focusses heavily on the suitability of the design for supporting quality requirements [CKK02, Bas06]. Assessment of the suitability of an architecture to deliver on quality requirements posses several challenges to the architecture team. As has been noted, quality requirements are frequently poorly specified and overly broad. The responsibility of requirements elicitation then falls on the architecture team to determine exactly which of these are most important, the scope of the requirement, the specifications of the requirement and acceptable tradeoffs among conflicting quality requirements.

Architecture-centric project management, and specifically ATAM, allows all stakeholders in a project to have input in determining the importance of quality requirements. While some of these are explicitly accounted for in the COCOMO model (eg. reusability, reliability, time and space constraints) most other qualities are not. Issues such as maintainability, subsetability, security, usability and portability are not directly accounted for in COCOMO, and will often be overlooked by the design and production team unless other stakeholders are given opportunity for input. This "explicit evaluation of the architecture of software systems with respect to the quality requirements will minimize the risk of building a system that fails to meet its quality requirements and consequently decrease the cost of system development" [Bos00].

In summary, the architecture-centric project management process provides the necessary components to apply risk management analysis while accounting for quality requirements. This includes risk identification, by ensuring input from all stakeholders, or at least a representative sample of stakeholders, is given during the architectural analysis stage. These quality risks are then prioritized by consensus of stakeholders, giving the project manager a first indication of the consequences of this quality not being met, and of the payoff in meeting the quality requirement. The third necessary component for risk evaluation is the likelihood of a risk occurring. Section 3.3.1 examines how the ATAM method elicits input from developers in order to gauge the probability of meeting given quality requirement under the proposed architecture.

5.2.1 Research Toward Formalizing Quality Requirements at the Architecture Level

Subramanian and Chung [SC05] conducted research exploring an architecture centered approach and focussed on incorporating the quality requirement of adaptability into the architecture. They sought to answer two questions:

- If all constituent components of an architecture possess a given non-functional requirement, will the system as a whole satisfy the requirement?
- What is the minimum number of components in an architecture that must possess a certain non-functional requirement, in order for the system as a whole to display that requirement.

By creating a goal interdependency graph with logical `AND` and `OR` relationships between components in order to maintain adaptability of the system, they examined how scenarios where varying numbers of constituent components of the architecture possessed this quality. A logical `AND` relationship here shows a case where system adaptability can be maintained only if all constituent components possess that quality. A logical `OR` relationship models the case where the quality can be maintained in the entire system, given that at least one of a subset of constituent components maintain the property. The graph takes on exponential degrees of complexity as these relationships are combined among components. While their research presents a way to identify this question, they lack real world architectures to test. This ability to identify the components necessary to maintain a quality attribute in the system allows for a significant reduction of risk at the architectural level. If a system's failure to meet adaptability requirements is identified at the architectural level, the expense of rework at later stages of the life-cycle can be avoided. That is, the added knowledge gained from identifying quality risks at the architectural stage reduces the uncertainty present ultimately leading to a reduction of risk, *at the point of architecture.*

5.3 Other Benefits of the Architecture-Centric Approach

5.3.1 Stakeholder Communication

Fairbanks et al. discovered that using architectural graphical models was an effective tool for bridging the communication gap between stakeholders [FBD06]. They found that parties from the business

domain, after being presented with basic UML models were able to participate in a planning in a relevant way. The UML provided a common notation in which stakeholders were able to properly *ground* the communication. This played an important role in making architecture decisions that fall on the boundary between technical issues and business issues. They present an example involving a high demand for data feeds on a shared repository. The stakeholders had two possible solutions to the problem, one requiring a technical solution by decentralizing the data, the other with a business solution involving a change to work practices. Both solutions had varying degrees of impact on different stakeholders, but the researchers discovered that the architectural UML models they employed provided a sufficient common reference for resolving the debate.

The importance of the architecture providing a communication mechanism to all project stakeholders cannot be understated. The artifacts produced during this process provides information which, when done correctly, addresses the concerns of all involved. Paulish specifically refers to the four separate architectures for this specific purpose. Clements et. al. suggest that one of the largest benefits of the architecture review process is “getting all the stakeholders in the same room at the same time” [CKK02]. Their work builds on the case studies used in developing the Software Engineering Institute’s ATAM process. The SEI researchers noted that “a group dynamic emerges in which stakeholders see each other as all wanting the same thing: a successful system” [CKK02]. Whereas before, their goals may have been in conflict with each other, now they are able to explain their goals and motivations so that they begin to understand each other. Increased levels of communication suggest that stakeholders can ensure at this early stage, that their concerns are addressed. Once again, by identifying concerns at the architectural stage, the uncertainty which leads to high risk areas is alleviated, allowing the risk to be reduced

5.3.2 Resource Assignment

Architecture first allows project managers to properly assign resources to projects. Projects with insufficient resources initially assigned account for a large amount of schedule overrun for the system [Bro95]. This problem becomes compounded when managers attempt to add additional engineers to the project once signs of overrun are present. As Brooks details, this actually has a tendency to increase the overall development time by adding “ramp-up time” and complexity to the communication process. This suggests that taking the architecture first approach prior to allocation of resources has potential for a significant impact in scheduling precision, ultimately resulting in a

reduction in risk. The Siemens researchers also cited success in using the architecture to finalize the composition of component teams. Paulish suggests that the identified top level components translate naturally into these teams of software engineers. Also, as noted in Section 5.1, this process will identify deficiencies in team experience, in some cases allowing the manager to add expertise to the project at an early stage in the development. Hence, by identifying the resources needed to not only implement the functional requirements, but also the quality requirements expressed by all stakeholders at an early stage, the project manager is able to ensure the resources are in place at an early stage. The knowledge gained by this ability to assign resources at the architectural stage again alleviates the uncertainty in the project, thus reducing risk.

5.3.3 The Project Schedule

Making a better schedule is cited by Paulish as a benefit to the architecture-centric approach. Brooks notes that dependencies among components leads to schedule overrun, as complete testing of a component cannot be complete until other components with which it is coupled are implemented [Bro95]. The dependencies identified among architectural components allows the manager to set checkpoints which account for these dependencies, leading to a more realistic and manageable schedule. Those components on which a high number of others depend can be developed and tested first, thus avoiding some of the delays associated with testing interdependencies. Royce and Keil suggest that implementing proper and reasonable checkpoints of milestones in the development of a large software project is a driving success factor [Roy98, KCLS98]. Based on this, the ability of architecture to aid the project manager in setting a reasonable schedule (and associated deliverables such as a work breakdown structure, software development plan) is another driving factor in helping ensure risks are mitigated over the life-cycle.

5.4 Ending Thoughts

5.4.1 Why Architecture?

This section addresses the issue of why there exists this large decrease in risk at the point of architecture. Much research has been conducted and is ongoing in an attempt to mitigate risk at the point of requirements. Indeed, considerable work is underway to use capability engineering to better

formulate the mapping between needs and requirements such that capabilities are produced which are change tolerant. These provide a natural transition to architecture components which can then be allocated to the relevant resources [RAB07]. While this work has large potential to produce systems with high conceptual integrity and incorporate quality attributes (eg. maintainability), research suggests that at the point of requirements there is not enough knowledge present to perform reliable effort estimates. Further, prior to developing the architecture of the system, decisions regarding top level components, frameworks and languages have yet to be made, thus removing the ability to assign resources to components in a knowledgable manner. At this point, the project manager is lacking the ability to set proper milestones and develop the work breakdown structure. In summary, this research suggests that the manager simply does not have enough knowledge to mitigate a substantial amount of risk.

Given that not enough knowledge is present at the requirements stage to mitigate substantial risk, the next question is whether or not a more appropriate phase of the life-cycle for this decrease in risk is at the conclusion of low-level design. Once low-level design is complete, the system is ready for implementation. Decisions have been made regarding not only major hardware and software components, frameworks and languages, but also data structures, algorithms and very fine grained details of the system. At this phase in the life-cycle, the project has exceeded the point at which the manager is in a position to mitigate some substantial risks. As the system is in a state to begin coding, if a need is seen for an experienced developer for a certain niche, the time pressure (and consequently, the cost) of obtaining that developer grows significantly. Of course, this principle applies to resources other than engineers also, such as computing resources.

5.4.2 Conclusion

This chapter identifies the factors involved with project estimation which are bounded by the process of architecture-centric project management. Based on the ability to more closely estimate the effort required to develop the software system, the risk of completing on schedule and on budget is greatly reduced. Similarly, through the architecture-centric management process, quality risks, not accounted for in traditional management practices are incorporated in the planning and risk evaluation. Lastly, Chapter 5 examines the reasons to support the premise that the largest drop off in risk occurs at the point of architecture, enumerating the knowledge lacking at the point of requirements and discussing the substantial effort involved in low-level design, which, while greatly

identifying and quantifying risks, occurs too late in life-cycle for the project manager to adjust plans to mitigate these risks.

Chapter 6

Conclusions and Future Work

6.1 Observations

Proponents of architecture have long cited its benefits of including consideration of quality requirements early in the design phase. Research of Siemens and the Software Engineering Institute also claims benefits to project management by providing the manager a framework for making decisions regarding scheduling and resource allocation. The benefits of this approach have been documented well in terms of software engineering theory and some cases studies. There has yet to be published many well documented case studies of large projects, especially viewed in comparison with large projects not employing an architecture-centric approach.

For small projects, this is too expensive due to the high overhead associated with formal estimation models and risk analysis techniques. Further, this cost grows significantly when faced with the decision between multiple architectures, each with numerous alternatives. As projects increase in size, however, the failure rate increases significantly [Jon98] and an early investment into an architecture-centric approach would seem to have potential to largely mitigate several of the risks associated with them. However, for large projects whose life-cycle spans years and whose budget is measured in millions of dollars, potential exists for management to implement an architecture-centric approach.

6.2 Conclusions

To review, Charette's steps in risk evaluation:

1. Identification of potential risk events through a structured and consistent method
2. Estimate the magnitude of each risk and its consequences, and the creation of options
3. Evaluate the consequences of risk [Cha89]

Steps one and two occur in architecture-centric management through the architecture design and evaluation process. During design, risks to project completion are identified by the architects. During architecture evaluation, the magnitude of each risk (ie. the importance it has to the project), the likelihood of occurrence and the consequence of the risk occurring are estimated. The accumulation of past work suggests that at the point of architecture, project managers are able to perform this risk evaluation strategy with more impact than in prior or successive life-cycle phases.

Chapter 5 examines the risks that are either alleviated or tightly bounded through the development of the architecture, namely estimation risks and risks associated with not meeting quality requirements. These risks are known to be poorly accounted for in traditional project management. Knowing that early life-cycle size and effort estimates have been established to be highly unreliable, this work enumerates how effort estimation parameters become more tightly bounded through the architecting process. Further, architectural evaluation as detailed in Section 3.3.1 provides a structured process to identify the quality risks from all stakeholders which are frequently not addressed.

A compilation of prior work suggests that at this point of architecture, the significant decrease in risk as a result of gained knowledge occurs. The work of Siemens practitioners and the SEI researchers specifically points to the ability to appropriately allocate and assign resources at an early stage. The accuracy gained by effort estimates suggests that at the point of architecture, enough information is known for this to occur. That is, major components, frameworks, languages and interfaces have been designed, so at this juncture, the project manager is aware of what needs exist to carry out low-level design and implementation. Further, experience from numerous project failures (see Section 1.1) suggests that discovering needs too late in the life-cycle, causes rework, added expense. That is, the lack of knowledge up to this point manifests itself, suggesting a high amount of risk present if quality factors and resource needs are not identified sooner.

Chapter 4 presented formalized and widely accepted methods of estimating and evaluating the risks associated with large software projects. These techniques can be used by project managers to and future researchers to prove that architecture-centric project management provides a way of measuring and mitigating estimation and quality risks. Notably, Section 4.3.2 examined the use of decision trees as a tool for evaluating alternatives, their expected values and risks associated with them. Software development, especially associated with large projects, will allays have substantial risk. These risks include failure to meet schedule, failure to meet budget, and failure to deliver on functionality or quality requirements. However, with solid software engineering practices, notably around the point of architecture, this work suggests that many of these risks can be mitigated and managed in a way to allow for a higher amount of projects to be successful.

Contributions of this research include providing a framework with which to consider the impact of architecture-centric management on reducing risk in large projects. Specifically,

- Mapping specific points of knowledge gained through the architecting process to the COCOMO II parametric effort estimation model
- Enumerating documented benefits (eg. providing communication mechanism, involvement of all stakeholders) of architecture-centric management not reflected in parametric models which have potential to significantly lower risk
- Suggesting a plan to prove the benefits of architecture first management to encourage widespread adoption
- Presenting evidence that prior to architecture, there is not enough knowledge present to substantially reduce estimation risk
- Discussing, from a management point of view, that scheduling which occurs in life-cycle phases subsequent to architecture is too late to allow optimal resource allocation

6.3 Future Work

6.3.1 Proving the Benefits of Architecture-Centric Project Management

In some cases, stakeholders openly question the investment of time into architecture development, rather than beginning coding, even on large projects [FBD06]. Currently, managers and consultants

offer what are essentially free estimates based on rules of thumb (see Section 4.2.1). Despite numerous amounts of evidence to the inaccuracy of manual methods, they remain the most widespread in practice [Jon98]. Further, even in cases where estimation models are used, investors are reluctant to enter into a full architecture-centric approach, oftentimes in favor of speeding up development [Fai03, BNWZ07]. This hesitation for widespread adoption may be mitigated by published work proving the success, rather than current literature which offers mainly best practices, isolated success stories and in some cases analysis based on a level of detail normally associated with low-level design metrics, not architecture. In order to prove this idea, this section presents an outline for how one would conduct an experiment with cooperation of a software producing organization involved with large projects.

Significant knowledge can be gained by examining the artifacts of architectural knowledge and design from past projects. Researchers can use the architectural design documents in order to conduct estimates using COCOMO II's post-architectural model. The results of these estimates can then be compared to actual effort expended on the project. Based on these comparisons, researchers would be able verify that at the point of architecture, effort estimates are predicted with a high degree of precision. Further, by comparisons of these estimates to actual outcomes, research can assess the ability of an architecture-centric approach to identify potential problems, or even doomed-to-failure projects at this early stage.

Estimation risks can be measured in a straight-forward manner at multiple times during the life-cycle. COCOMO II presents three models corresponding to various stages in the software life-cycle. Researchers would conduct estimation of several projects at each of these phases:

1. The prototyping model conducted from early user-requirements (ie. customer needs) documentation

2. The early design model after the first iteration of architecture with early design documentation and requirements specification

3. The post-architectural model following completion of the architecture

Following the life-cycle of the project to release, the accuracy of each of these models can be compared to the actual effort expended on the project. Once this information is known, the estimation risk can be calculated given the (1) the likelihood of each technique, corresponding to

life-cycle phase, accurately predicting the effort within a given tolerance and (2) the consequence of the overrun, measured in schedule slippage and dollars as a function of programmer-months.

Following the calculation of risk based on the three estimation models, researchers can calculate what is known as the *expected value of information* (EVOI) [Cha89]. This figure represents the what the information gained through the architecting process is worth to the project manager. This calculation begins by calculating the expected value of an given estimate. This is, essentially, the sum of the product of the probabilities of each outcome, in this case where the project finishes with respect to its schedule, and the cost or payoff of each alternative (see Section 4.3.1 for more information). The expected value of the information is the difference of the expected value and the cost of obtaining the information, in this case the cost of developing the architecture.

By repeating this study over several projects, a compilation of data concerning the expected value of information can be accumulated. If, as expected, this value consistently can be shown to be positive (the cost of developing the architecture is less than than the expected value of estimation), more widespread acceptance of an architecture-centric approach is likely to occur. It is possible even that there exists an economy related to developing multiple architectures. That is, prior to a decision for implementation, the risks can be explored with each architectural alternative and still be measured as a financially favorable outcome.

This experiment details the calculation of estimation risk only accounting for functional requirements. A more interesting point may even exist whereas the quality requirements are considered, and some measurement of the final projects quality in terms of usability, portability, maintainability, etc... be conducted and compared with those of projects which failed to employ an architecture evaluation such as ATAM. To prove the benefits of this approach with respect to quality, it is possible to measure the effort spent on rework, due to quality issues being addressed late in the life-cycle, in comparison with the cost spent developing the architecture and cost spent on subsequent architecture evaluation. Assuming that a representative group of stakeholders can be assembled to participate, the ability of the architecture to accommodate quality requirements can be assessed. Based on this assessment, again, results from ATAM can be then compared to the ability of the developed system to deliver on quality requirements. Large potential exists here by studying projects which failed to deliver quality requirements to verify that indeed, this would be noted by performing architecture analysis.

Verifying results of COCOMO II estimation models and the Architecture Tradeoff Analysis

Method against known performance of successful and failed projects, has potential to reap large amounts of data, specifically to verify the inflection point around architecture. If, as expected, the process of developing and evaluating the architecture prior to resource allocation allows for accurate predictions with regard to effort and the ability of the proposed system to incorporate quality attributes, the hypothesis is proved.

6.3.2 Using Architecture-Centric Estimation to Define Architecture

As an interesting side-note to this research, note that evidence presented from numerous industry and academic research projects with size and estimation suggests that the process of evaluating risk may help define the point at which *architecture* is separated from *low-level design*. This suggests that this delineation exists where the software project manager believes to have enough granularity of detail in order to bound the estimates of risk tightly enough in order to arrive at a decision. That is, the process of risk evaluation can be considered as a method of arriving at an answer to the question of "what is an a software architecture" that was discussed in Section 2.5.3. In many cases, a course level of detail is sufficient to provide accurate effort estimates and assess the quality risks associated with a project. In other cases, such as safety-critical systems, real-time embedded systems, or those for which the platform, language or other tools are relatively new to the project's engineering team, may require a much finer grained level of detail. This belief is captured graphically in Figure 2.3.

Following more research compiling data from architecture first approaches, it is possible to develop heuristics suggesting what levels of detail are required to obtain the desired information. This level of detail could be measured by number of each types of diagrams(class, sequence, timing, etc..), amount of information present on the diagram(type of dependency, information passing mechanism, class attributes, methods, etc..) or number and types of tables and charts reflecting traceability and mapping between diagrams. This level of detail could be grouped by business domain and computer science domain, and precedentedness of the project, platform, language or other tool to the development team. Guidelines suggesting a level of necessary detail could prove to be useful for accurate estimation and stakeholder understanding of the project.

REFERENCES

[ADSJ01] Bente Anda, Hege Dreiem, Dag I. K. Sjøberg, and Magne Jørgensen. Estimating software development effort based on use cases-experiences from industry. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 487–502, London, UK, 2001. Springer-Verlag.

[Alk06] Mohammad A Alkandari. Investigation into cultural aspects, personality, and roles of software project team configuration. Master's thesis, Virginia Tech, 2006.

[AV02] Tom Addison and Seema Vallabh. Controlling software project risks: an empirical study of methods used by experienced project managers. In *SAICSIT '02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 128–140, , Republic of South Africa, 2002. South African Institute for Computer Scientists and Information Technologists.

[BAC00a] Barry Boehm, Chris Abts, and Sunita Chulani. Software development cost estimation approaches a survey. *Ann. Softw. Eng.*, 10(1-4):177–205, 2000.

[BAC00b] Barry W. Boehm, Chris Abts, and Sunita Chulani. Software development cost estimation approaches - a survey. *Ann. Software Eng.*, 10:177–205, 2000.

[Bal06] Osman Balci. Software development life cycle waterfall model. In *Software Engineering CS 3204*, Lecture Notes in Computer Science, 2006.

[Bar06] Andrew Bartels. http://www.forrester.com/Research/Document/0,7211,40451,00.html, 2006.

[Bas06] Len Bass. Principles for designing software architecture to achieve quality attribute requirements. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, page 2, Washington, DC, USA, 2006. IEEE Computer Society.

[BCH+95] Barry Boehm, Bradford Clark, Ellis Horowitz, Richard Shelby, and Chris Westland. An Overview of the COCOMO 2.0 Software Cost Model. In *Software Technology Conference*, April 1995.

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture In Practice*. Addison-Wesley, 1998.

[BM97] J. Bosch and P. Molin. Software architecture design: Evaluation and transformation, 1997.

[BNWZ07] Len Bass, Robert Nord, William Wood, and David Zubrow. Risk themes discovered through architecture evaluations. *wicsa*, 0:1, 2007.

[Boe84] Barry W. Boehm. Software engineering economics. *IEEE Trans. Software Eng.*, 10(1):4–21, 1984.

[Bos97] J. Bosch. Specifying frameworks and design patterns as architectural fragments, 1997.

[Bos00] Jan Bosch. *Design and Use of Software Architectures*. Pearson, 2000.

[BP88a] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Trans. Softw. Eng.*, 14(10):1462–1477, 1988.

[BP88b] Barry W. Boehm and Philip N. Papaccio. Understanding and controlling software costs. *IEEE Trans. Software Eng.*, 14(10):1462–1477, 1988.

[BR98] J. Baragry and K. Reed. Why is it so hard to define software architecture? In *APSEC '98: Proceedings of the Fifth Asia Pacific Software Engineering Conference*, page 28, Washington, DC, USA, 1998. IEEE Computer Society.

[Bro95] Frederick P. Brooks. *The Mythical Man-Month*. Addison Wesley, 1995.

[Car05] Edward R. Carroll. Estimating software based on use case points. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 257–265, New York, NY, USA, 2005. ACM Press.

[Cha89] Robert N. Charette. *Software engineering risk analysis and management*. McGraw-Hill, Inc., New York, NY, USA, 1989.

[CHA94] `http://www.standishgroup.com/sample_research/chaos_$1994_1$.php`, 1994.

[Cha05] Robert N. Charette. Why software fails. *IEEE Spectrum*, 42(9):42–49, September 2005.

[CKK02] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architecture: Methods and case studies*. Addison-Wesley, 2002.

[CR99] Elliot Chikofsky and Howard A. Rubin. Using metrics to justify investment in it. *IT Professional*, 1(2):75–77, 1999.

[CS] M. Carbone and G. Santucci. "fast & serious: a uml based metric for effort estimation". In *Proceedings of the 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*.

[DK75] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM Press.

[Dob02] Ernst-Erich Doberkat. Pipes and filters: Modelling a software architecture through relations. Memo 123, Lehrstuhl Software-Technologie, University of Dortmund, June 2002.

[DvdHT05] Eric M. Dashofy, Andrø; van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Methodol.*, 14(2):199–245, 2005.

[Fai03] George Fairbanks. Why can't they create architecture models like "developer x"?: an experience report. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 548–552, Washington, DC, USA, 2003. IEEE Computer Society.

[Fai06] Richard E. (Dick) Fairley. The influence of cocomo on software engineering education and training. In *CSEET '06: Proceedings of the 19th Conference on Software Engineering Education & Training (CSEET'06)*, pages 193–200, Washington, DC, USA, 2006. IEEE Computer Society.

[FBD06] George Fairbanks, Kevin Bierhoff, and Desmond D'Souza. Software architecture at a large financial firm. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 815–823, New York, NY, USA, 2006. ACM Press.

[FDH06] Yujian Fu, Zhijiang Dong, and Xudong He. Formalizing and validating uml architecture description of web systems. In *ICWE '06: Workshop proceedings of the sixth international conference on Web engineering*, page 8, New York, NY, USA, 2006. ACM Press.

[GB98] Håkan Grahn and Jan Bosch. Some initial performance characteristics of three architectural styles. In *WOSP '98: Proceedings of the 1st international workshop on Software and performance*, pages 197–198, New York, NY, USA, 1998. ACM Press.

[Gla06] Robert L. Glass. The standish report: does it really describe a software crisis? *Commun. ACM*, 49(8):15–16, 2006.

[Har06] `http://www.infoq.com/articles/Interview-Johnson-Standish-CHAOS`, 2006.

[HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied software architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[HPB05] James D. Herbsleb, Daniel J. Paulish, and Matthew Bass. Global software development at siemens: experience from nine projects. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 524–533, 2005.

[Int07a] International Function Point Users Group. *http://www.ifpug.org/about/*, 2007.

[Int07b] International Function Point Users Group. *Origins of Softare architecture study*, 2007.

[Jon95] Capers Jones. Backfiring: Converting lines-of-code to function points. *Computer*, 28(11):87–88, 1995.

[Jon98] T. Capers Jones. *Estimating software costs.* McGraw-Hill, Inc., Hightstown, NJ, USA, 1998.

[KCLS98] Mark Keil, Paul E. Cule, Kalle Lyytinen, and Roy C. Schmidt. A framework for identifying software project risks. *Commun. ACM*, 41(11):76–83, 1998.

[Kru02] Philippe Kruchten. Tutorial: introduction to the rational unified process®. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 703–703, New York, NY, USA, 2002. ACM Press.

[MA02] J. Susan Milton and Jesse C. Arnold. *Introduction to Probability and Statistics: Principles and Applications for Engineering and the Computing Sciences.* McGraw-Hill Higher Education, 2002.

[Mac01] Leszek A. Maciaszek. *Requirements analysis and system design: developing information systems with UML.* Addison-Wesley Longman Ltd., Essex, UK, UK, 2001.

[MAC05a] Parastoo Mohagheghi, Bente Anda, and Reidar Conradi. Effort estimation of use cases for incremental large-scale software development. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 303–311, 2005.

[MAC05b] Parastoo Mohagheghi, Bente Anda, and Reidar Conradi. Effort estimation of use cases for incremental large-scale software development. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 303–311, 2005.

[Met05] Methods & Tools. *http://www.methodsandtools.com/mt/download.php?spring05*, 2005.

[MIT02] MIT OpenCourseware. *http://www.core.org.cn/OCW_CN/Civil-and-Environmental-Engineering/1-264JDatabase–Internet–and-Systems-Integration-TechnologiesFall2002/LectureNotes/index.htm*, 2002.

[MJ03] K. Moløkken and M. Jørgensen. A review of software surveys on software effort estimation. In *ISESE 2003: International Symposium on Empirical Software Engineering*, pages 223–230, 2003.

[MOT+00] Nenad Medvidovic, Peyman Oreizy, Richard N. Taylor, Rohit Khare, and Michael Guntersdorfer. An architecture-centered approach to software environment integration. Memo USC-CSE-00-516, Center for Software Engineering, University of Southern California, March 2000.

[NPSS01] Robert L. Nord, Daniel J. Paulish, Robert W. Schwanke, and Dilip Soni. Software architecture in a changing world: developing design strategies that anticipate change. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 309–310, New York, NY, USA, 2001. ACM Press.

[NT06] Robert L. Nord and James E. Tomayko. Software architecture-centric methods and agile development. *IEEE Software*, 23(2):47–53, 2006.

[Oqu04] Flavio Oquendo. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, 29(3):1–14, 2004.

[Oqu06a] Flavio Oquendo. Formally modelling software architectures with the uml 2.0 profile for π-adl. *SIGSOFT Softw. Eng. Notes*, 31(1):1–13, 2006.

[Oqu06b] Flavio Oquendo. π-method: a model-driven formal method for architecture-centric software engineering. *SIGSOFT Softw. Eng. Notes*, 31(3):1–13, 2006.

[Par92] R. Park. Software size measurement: A framework for counting source statements, 1992.

[Pau02] Daniel J. Paulish. *Architecture-Centric Software Project Management: A Practical Guide*. Pearson, 2002.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[RAB07] Ramya Ravichandar, James D. Arthur, and Shawn A. Bohner. Capabilities engineering: Constructing change-tolerant systems. *hicss*, 0:278b, 2007.

[Rei00a] Donald J. Reifer. Web development: Estimating quick-to-market software. *IEEE Softw.*, 17(6):57–64, 2000.

[Rei00b] Donald J. Reifer. Web development: Estimating quick-to-market software. *IEEE Softw.*, 17(6):57–64, 2000.

[RKJ04] Sunghwan Roh, Kyungrae Kim, and Taewoong Jeon. Architecture modeling language based on uml2.0. In *APSEC*, pages 663–669, 2004.

[Roy98] Walker Royce. *Software Project Management: A Unified Framework*. Addison-Wesley, 1998.

[SC05] Nary Subramanian and Lawrence Chung. Relationship between the whole of software architecture and its parts: An nfr perspective. In *SNPD-SAWN '05: Proceedings of the Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks (SNPD/SAWN'05)*, pages 164–169, Washington, DC, USA, 2005. IEEE Computer Society.

[SC06] Mary Shaw and Paul Clements. The golden age of software architecture. *IEEE Softw.*, 23(2):31–39, 2006.

[SG96] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[SNH95] Dilip Soni, Robert L. Nord, and Christine Hofmeister. Software architecture in industrial applications. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 196–207, New York, NY, USA, 1995. ACM Press.

[Sof06] Software Engineering Institute. *http://www.sei.cmu.edu/architecture/ata_method.html*, 2006.

[Som04] Ian Sommerville. *Software Engineering*. Pearson, seventh edition, 2004.

[SX03] Petri Selonen and Jianli Xu. Validating uml models against architectural profiles. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 58–67, New York, NY, USA, 2003. ACM Press.

[VS94] D. H. Von Seggern. *PHB practical handbook of curve design and generation*. CRC Press, 1994.

[Zus97] Horst Zuse. *A Framework of Software Measurement*. Walter de Gruyter & Co., Hawthorne, NJ, USA, 1997.

VITA

Troy Steven Henry
2024 Gillwell Lane
Fuquay-Varina, NC 27526-5342

Education Despite his best efforts to the contrary, Troy graduated Blacksburg High School in 1988.

Following this near escape, he embarked on a collegiate education at Virginia Tech. However, after a series of self-termed “extremely poor choices” he left college after one year, on academic probation. In order to convince himself of the necessity of a college education, he pursued a sixteen year career in quick-service restaurant management. In the Summer of 2001, Troy returned to Virginia Tech on a part-time basis, renewed in his determination to obtain a degree in computer science. While in school part-time over five years, Troy managed to progress from freshmen to senior, and gain admission to Virginia Tech’s 5 Year BS/MS program. As it would turn out, Mr. Henry’s academic failures during the 1980’s served him well by priming his transcript so that he could earn the George Gorsline Memorial Scholarship for “Most Improved Student in Computer Science.” This scholarship allowed him to leave his full-time restaurant job and attend to his education. Troy completed his bachelors degree in Computer Science, with a minor in Mathematics in August of 2006, graduating Summa Cum Laude and winning Computer Science Department’s “Most Outstanding Senior” award. In May of 2007 he finished his thesis in software engineering, attempting to integrate his experience in management with his education in computer science.

Professional Accomplishments Troy was named Tacoma, Inc.’s “Manager of the Year” in 2000

and was the only manager in Tacoma’s 19 year history to have his store named franchise “Restaurant of the Year” on three occasions. He received numerous other recognition and achievement awards over his sixteen year Taco Bell career, but is most proud of his ability to win two of the “Restaurant of the Year” awards while also attending Virginia Tech on a part-time basis. Troy has accepted a position as Software Design Engineer with Plexus technology group (`http://www.plexus.com/`) in Raleigh, NC. He hopes to integrate his technical skills and management experience at Plexus for a highly satisfying and rewarding career.

Personal Troy is married to a wonderful, patient and highly understanding woman, Peggy Lowe. He

continues to credit Peggy with the majority of his personal, academic and professional success. He

is the proud father of Justin Lowe, USAF, and Kayla Henry, whose early academic accomplishments have far exceeded her father's. Troy and his family are members of the Unity Church (`www.unity.org`), a spiritual community he has found to be extremely supportive of his goals, and whose values are aligned with his spiritual and logical beliefs. Upon completion of his academic goals, he hopes to resume activities such as running, hiking and camping.