APPLICATION OF SOFTWARE ENGINEERING BEST PRACTICIES AND PRINCIPLES TO SMALL DEVELOPMENT TEAMS

by

MILES HENRY PHILLIPS

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2006

ACKNOWLEDGEMENTS

I would like to acknowledge my supervising professor Mr. David Levine for his guidance throughout the development of this thesis. He provided good feedback and encouragement to make this thesis possible. I also appreciate Dr, Sharma Chakravarthy and Dr. Roger Walter for their participation on my committee.

I would be amiss to not mention my gratefulness to my wife Kristi and my children Emily, Elizabeth, Ellen and Benjamin, for their loving support during my pursuit of higher education. I am forever indebted to my Lord Jesus Christ for the inspiration, strength and mercy given to me. It is for His glory and honor that I have pursued this degree.

April 3, 2006

ABSTRACT

APPLICATION OF SOFTWARE ENGINEERING

BEST PRACTICES AND PRINCIPLES

TO SMALL DEVELOPMENT TEAMS

Publication No. _____

Miles Phillips, M. S.

The University of Texas at Arlington, 2006

Supervising Professor: David Levine

The motivation of this thesis comes from the professional experience of the author. Having worked with very small software development teams in various capacities, he realized that significant improvements could be achieved by the application of modern software engineering practices and principles. This paper is the result of researching how the principles and practices promoted by the leading software development authors can be applied to the small development team as they transition from “programming in the small” to “programming in the medium.”

This paper investigates the best practices and principles in various case studies with the objective to clearly define how the application of these practices and principles contributed to a successful software project. Several chapters are dedicated to reviewing techniques applicable to each of the phases of the software lifecycle with examples of each technique. The paper culminates with the author's recommendation of principles and practices for the small development team which can and should be used to improve the quality and overall time of the software development lifecycle of a small project. Each practice used in these case studies is evaluated for practical use in a small project, discussing the advantages and disadvantages of each tool while also exposing some reasons why trained software engineers often neglect these practices. The paper concludes with suggestions from the author of appropriate application of the practices and principles to small development teams.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Thousands of programmers have produced useful products without consciously applying any software engineering principles or without producing any written requirements, design or testing plans. These professionals often balk at the "overhead" encouraged by software engineering courses and literature, considering such time consuming "extra" activities a luxury or "red tape". Many of the programmers are absolutely correct. Certainly, a programmer can be professionally successful without adhering to the modern software engineering principles and techniques. However successful the product was or however quickly the product was produced cannot be the only measure to verify that software engineering practices do not need to be followed, even for the small projects. Issues like maintenance ease of revision, reuse of components, duration of testing and integration, and difficulty of training all must be considered. When measuring the success of the project, the entire development cycle must be considered. The old adage "there's always room for improvement" holds true for software engineering. Software engineering addresses software from the "big picture" perspective. A small development team is often tempted to take the informal or undisciplined approach especially if the project is considered "small" or is a modification to an existing system due to a variety of reasons. While the product may be initially delivered quicker in an undisciplined manner, the team must consider the

downstream effects. Did the lack of requirements lead to another cycle of development based on customer dissatisfaction? How many critical bugs were found after product delivery? How much of the system's functionality will be repeated in future projects? How long does it take for a developer in the team to maintain another developer's code? The small development team should evaluate the ideas conveyed in this paper since the industry has a poor record of completed projects. The Standish Group reported in 1994 that only 16.2% of software engineering project finish on time and on budget. [STANDISH1994]. This group reported in *The Chaos Report* an increase in 2003 of 34% of projects finishing on time and on budget. While this is a significant improvement, it still remains that 66% of all projects fail to meet this basic objective. [STANDISH 2003] Granted this is just one metric, but it illustrates the need for improvement in the development of software projects. The author of this paper is convinced that by applying software engineering best practices and techniques to some degree, any small development team can improve, drastically in some cases, its ability to effectively delivery software products.

The author of this paper writes from the perspective of over five years as a commercial software developer and ten years as a system administrator, and "utility" programmer. He is an example of a software professional who has achieved some success in an "unstructured" development environment. Often the software engineering community disregards this approach at any time. However, this author has had years of successful experience researching new ways to automate mapping application and producing utilities programs and prototypes. In these cases, it could be argued that the

unstructured approach is very efficient for rapid development for very small problems or proof of concepts. In hindsight however, there are some universal principles that would have improved his projects as they evolved into larger system. Having been employed in roles as a utility programmer for the majority of his professional, he has been productive without taking the time to formalize requirements, produce detail design, or many other software engineering activities he was trained to do. It seldom seemed practical to the situation. However, he transitioned several years ago into a software developer for a small development company. His first task was to research and develop a product that would manage the spatial aspects of the existing system into a database management system instead of the existing file based approach. He developed a prototype which after several iterations was accepted and then this prototype evolved into a product without rebuilding the system using the practices and principles mentioned in this paper. The evolved product has resulted in a disproportionately amount of time spent on integration and maintenance. When the opportunity to redesign the system presented itself, he was able to convince his management to expend the resources to apply the appropriate software engineering principles and practices to the project. The expected results are much shorter integration and maintenance cycles, better communication of functionality, more intuitive usability and reusable requirements, designs and components to provide a foundation for future projects. This event happened approximately the time that the author was selecting his thesis topic and therefore was of much interest to him. His experience is not an isolated occurrence. Many well meaning software developers for a

variety of reasons have neglected proven practices. The goal of this paper is to provide a concise evaluation of modern software engineering practices and principles.

The author is currently going through the pains of evolving from the practices of "programming in the small" to those appropriate for "programming in the medium". "Programming in the small" has been defined as the development of the class of programs that can be understood by one person, meaning that a developer can modify the code without unexpected consequences and without referencing outside documentation. [ZIMMER] It can further classified as a system that can be maintained effectively without documentation over a sustained period of time. Outside documentation may be needed to know what somebody else expects the code to do, but no other documentation is needed to understand how the code works unless it implements a technique new to the developer. While developers vary greatly in their ability to mentally hold the details of a system, the human mind is limited in what can be clearly retained. Obviously this "measure" is simply an attempt at determining to what extent the developer should apply the software engineering principles and practices. The answer is a scalable one, meaning that some are universal and would benefit a program of any size, such as coding standard, while others are practical for large complex system, or "programming in the large". The "large" class of programs is defined by systems that must be thoroughly documented in order for the system to be understood and is performed by larger groups of people or by smaller groups over longer time. It produces code that cannot be understood without a divide and conquer approach. With programming in the small, the emphasis is on clean code that can be

understood. With programming in the large, the emphasis is on partitioning the work into modules whose interactions are precisely specified. This requires careful planning and careful documentation. [ZIMMER] Obviously, "programming in the medium" would be the class of systems that are in between. Programming in the medium allows for the flexibility shared with the small project but is large enough to require stricter adherence to documentation and techniques. Many small to medium-sized system can be built with today's tools by one to two developers. Not only is the "size" or complexity of a project to be considered when determining what principles and practices to follow but also the risk factor of the project. With a low risk project, little management support or review is needed. Large systems need more structure and review, involve many customers and users and development occurs over a long period of time. [PFLEEGER pg. 32] The basic assumption of this paper is that developing software with some degree of a disciplined approach will always result in more stable, useful, correct products that are produced in a more efficient manner. One of the goals of this paper is to assist the small development team in recognizing the practices and principles that are "universal", ie. applicable to projects of all sizes as well as to introduce those that are useful to projects large enough to benefit from a more disciplined approach.

A software developer transitions into a software engineer when he or she adheres to software engineering principles and utilizes the techniques shared amongst the software engineering community. No matter the size of the project, some level of software engineering principles, practices and techniques can improve the development.

Software engineering has much to offer developers of all types. Many authors have written extensively on software engineering principles, methodologies, and techniques so there is no shortage of material on the subject. Any respectful computer science collegiate program includes several courses in solid software engineering on how to develop efficient software. With this level of maturity of the software development discipline, it would seem logical to conclude that trained software developers would follow these proven techniques and thus more times than not, successfully and efficiently produce stable and useful products. However, many developers, particularly "developers in the small", choose to not to follow the more disciplined approach, resulting in the longer term in faulty or unusable systems or missed deadlines or yet another one off product that does not allow the reuse any portion for future products. Reasons for the omission of best practices range from lack of education to being lazy to unsupportive management.. Also it takes discipline to follow a prescribed methodology and many developers want to start programming after being given a description of the problem. As stated in the opening paragraph, adhering to defined practices appears to be a waste of time to this mindset, though studies show that projects without adequate time given to each development activity has expensive repercussions later in the project. For example if not enough time is spent on gathering requirements for a "project in the medium", the result would most likely be a product that does not meet the need of the customer Developers prescribing to this undisciplined approach would not be considered software engineers, for their actions do not demonstrate a systemic, quantifiable approach to developing the product, which is at the core of the definition of

software engineering. [IEEE610 pg . 67] .Why does this negligence continue and in some cases be rewarded by management? Some legitimate reasons include

- Some programs are small enough can be developed efficiently by just "hacking it out". These products are often utility programs that can be easily revised and edited "on the fly" as bugs are found or the requirements for the tool changes. Programs that fall into this category would be fall in the "programming in the small" classification.
- The management is pressuring the developer to just get it done, therefore tempting well-meaning developers to "cut corners" to accomplish the task in the deadline given. These short cuts often lead to products that are riddle with bugs, therefore taking a grossly disproportionate amount of testing and "bug fixing" time or worst, sending out a faulty deliverable, causing loss of trust or respect for a company by its clients. Why do developers feel that taking short cuts will actual shorten the product cycle time? There is always the feeling that "everything is going to go right this time", even when it rarely does. This faulty feeling results in accepting an email for a requirements document or testing of the standard (or "happy path") case as the final system test. While occasionally these practices can be tolerated, they never pay off in the long run.
- Developer may have spent years of "programming in the small", building utility programs or performing research efforts, and then suddenly find that one of his/her projects have grown into a sustain system without the conscious application of today' best practices or to adhering to the proven

principles that guide a substantial software development effort. In such situations it is difficult to reverse the "hacker" trend without a conscious effort of the development team and the management.

This paper is written for the benefit of software developers that find themselves faced with medium sized development efforts with a small development team (two or three developers). The ad hoc approach to development cannot be effectively be used for "programming in the medium". The term "medium" is relative but to provide some metrics, this author considers a medium software project to contain 5,000 to 50,000 lines of code, 200 to 400 functions, and requiring more than one man month to complete. Such projects require the developer to consciously employ software engineering principles, practices and techniques in order to successfully develop the product, although not to the degree that "programming in the large" requires. Much of the software engineering literature is written for "programming in the large", leaving each development team to determine which techniques to incorporate and to what extent is practical for this project. Certainly each project would benefit from the application of modern software engineering principles, methodologies and techniques but to varying degrees. It is time-consuming for a small development team assimilate the vast information available and develop a well thought out and applicable approach to software development. The intent of this paper is to examine several of these tricks of the trade from the perspective of "programming in the medium", review how others have used them and to attempt to provide an example of how to implement a small

development project generated by efficient use of the best that the software engineering discipline has to offer today.

Software engineering practices and methods fall into one of two concerns: : managerial and technical. Applying project management (managerial) practices and techniques for the planning, scheduling and controlling of a development project are critical to the success of any medium to large project. Software projects benefit from implementing good, basic project management principles and practices. A host of software engineering professionals have developed solid project management approaches with ideas and guidelines borrowed from the broader project management community and much has been written on this managerial aspect of software engineering. These concerns are not the focus of this paper. Separating the managerial concerns from the technical permits a focused discussion on techniques that can be applied in a variety of managerial techniques. This paper focuses on the technical aspect of the trade, such as how to define requirements, create useable design documentation, and generate tests that validate the requirements. The ideas presented in this paper can, in some form, be applied to most any managerial methodology.

While the focus is not on the managerial side of software engineering, it is necessary to discuss the roles required for an effective software development team. One of the first steps to improving the technical aspects of a development team is to recognize every member's role (who is doing what) and how to efficiently carry out these roles. The responsibilities of the roles discussed below should be performed to some degree no matter what the size of the development team. If a project has a single

developer, then that person must assume all the roles while larger teams can divide the responsibilities as skills allow. The assumption is that if a role is poorly implemented or unconsciously carried out, then the byproducts of this neglect will negatively affect the project. For example, requirements for a project may be a simply a discussion at the break room or an email with a vague description of the system. Such requirements are impossible to reference in design documentation or in test plans therefore not providing the means for traceable requirements. In this case, the team member responsible for requirements should recognize this deficiency and prepare an adequate level of documentation.

Once the roles of a team are established, the development team must evaluate the skills of its members and determine if all roles can be assigned and adequately be performed by the team. Large development environments often have the luxury of choosing team members appropriate for a development effort while small development companies will often have to adapt or re-educate its developers to appropriately equip the development team for the project. It is important that the team size is optimal for the development effort. Doubling the size of a team will not half the development time. [FACTGURU] No matter the size of the team, it is important to consciously develop and maintain the efficient execution of each role. The roles are described in the table below.

Table 1.1 Software Development Team Roles [PFLEEGER pp. 25-27]

Role	Description
requirements analyst	Represents the activities of working with the customer to understand what the problem is and how a new or modified software system could provide a satisfactory solution to the problem. The person or persons in this role should have good communication skills as well as analytical skills so that they can define the project into tangible, concrete pieces accurately reflecting what the customer desires.
designer	Involves generating system level descriptions in such a way that a programmer can write lines of code which will implement what the requirements specify
tester	Involves working with the implementation team to verify that, as the system is build up by integrating its components, it works properly and according to the specifications.
trainers	Directs users on how to use the system efficiently.
maintenance	Consists of implementing bug fixes or enhancements
librarian	Includes preparing and storing artifacts used for the life of the system. These documents include requirements and design specifications, training manuals, test data and schedules as well as the software components.
configuration manager	Has the responsibility of maintaining correspondences among requirements, design, implementation and tests

Each member of the development team should realize his or her role or roles and coordinate together to efficiently develop the final product. Ideally, each team should include at least two people capable of performing each role, so that if some

member leaves the team or is absent, that role can continue. [FACTGURU] Many of these roles correspond to a particular develop phase which will be discussed later and are not required throughout the life of project and therefore could be a resource that is assigned to multiple projects. Neglect of any of these responsibilities will result in a less efficient use of the team's resources. Therefore each role must be developed to increase the maturity of a software development. The object of this paper is to provide a concise summary of effective principles, practices and techniques to equip the software developer to effectively fulfill his or her role or roles within the development team that finds itself "programming in the medium" and has the difficult task of determining a balance between "over engineering" the project versus not applying sufficient software engineering techniques to generate a quality product in an efficient manner.

In order to accomplish this objective, this paper has been organized in a manner to lead the reader to full understanding of the software engineering discipline in chapter two. Chapter three reviews current software engineering principles and practices and provides some comments on how to apply to "programming in the medium". Chapter four discussions some of the current standards available for the software engineering discipline. Chapter five introduces the case studies are referenced throughout the remaining chapters. Chapter six details various techniques that can be used in various phases of the software development lifecycle. Chapters seven through twelve describe the phases of the lifecycle and review techniques specific to that phase. Chapter

thirteen concludes the paper with the author's application of software engineering to the "programming in the medium".

CHAPTER 2

SOFTWARE ENGINEERING EXPLAINED

2.1 Definition of Software Engineering

Software engineering is formally defined by Institute of Electrical and Electronic Engineers (IEEE) as "1. the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, that is, the application of engineering to software. 2. The study of approaches as in 1." [IEEE610 pg . 67]. Alan Davis states "If software engineering is really an engineering discipline, it is the intelligent application of proven principles, techniques, languages and tools to the cost-effective creation and maintenance of software that satisfies users' needs. [DAVIS95 preface] The distinguishing factor between simply creating a software program and software engineering is in how the software is created. Software engineering is the conscious application of a disciplined approach as stated in the above definitions. Therefore a development project that does not follow "a systematic, disciplined, quantifiable approach" cannot be considered a software engineering project. This chapter describes software engineering by defining terminology and describing some common approaches in order to lay a foundation for the remaining chapters of this paper.

2.2 Software Engineering Terminology

Throughout the research for this paper, several terms were used in a variety of ways to describe and enumerated current thought and practice of the modern software engineering discipline. In order to consistently refer to these terms in this paper, it is necessary to provide an acceptable description of each.

Practices are simple the way that something is done. They are not implemented by technologies, but are conceived by humans. [WIKIPEDIA PRACTICE] "Practices are the evidence of values." [BECK pg. 14]

Principles are rules or norms that form the foundation for decisions and actions. They vary from guidelines in that guidelines are more suggestions but a principle are more edicts that suggest if a principle if violated, then the product will directly be negatively impacted. [DAVIS95 pg. 3] Principles bridge the gap between values and practices. They are domain specific guidelines. [BECK pg. 15]

Techniques are procedures that aid in developing software in performing some subset of a software development phase, thus enforcing some underlying principles. Many techniques result in documentation or are used to transform existing documentation into a product. [DAVIS95 pg. 3]

Languages are a set of primitive elements and set of rules that can be used to create more complex entities. Techniques create documents or products that are represented by some language. [DAVIS95 pg. 4]

Methodologies in the context of software engineering referred to "a codified set of recommended practices, sometimes accompanied by training materials, formal educational programs, worksheets, and diagramming tools. While these would be more accurately referred to as *methods*, the word *methodology* is a more grandiloquent." [WIKIPEDIA METHODOLOGY]

Tools are software programs used to enforce tool, support languages and techniques in order to more efficiently carry out some step of software engineering. [DAVIS95 pg. 4] Shari Pfleeger refers to a tool as "an instrument or automated system for accomplishing something in a better way ie. more accurate, efficient, productive or enhances quality." [PFLEEGER98] Many excellent software engineering tools are available to assist software development. Tools cover the full gambit of software engineering activities, from project management to system modeling tools to integrated development environments to software configuration management. Tools include such a text editor or, database management system, testing support, and programming environments. It is not the intent of this paper to review such programs but to focus on the techniques that these tools automate. If a development team finds a particular technique helpful in their environment, then the team should at that point start looking for a tool to automate that technique. It is highly recommended that a technique be utilized manually before spending the time and money on procuring an automated tool for the simple reason if it is not useful manually then it will most likely not be useful automated. Many procured tools quickly become "shelfware" after the novelty wears off.

Alan M. Davis depicts the relationship between principles, techniques, languages and tools in the following diagram.

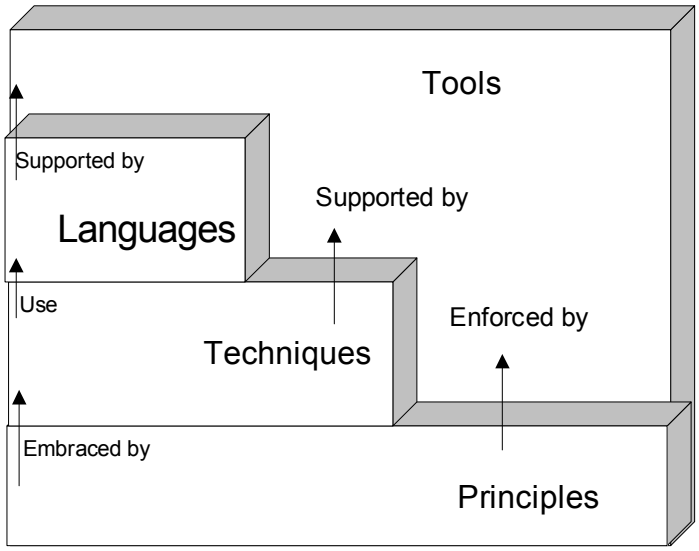

Figure 2.1 Relationship of Principles and Techniques [DAVIS95 pg. 4]

A process is a series of steps involving activities, constraints and resources that produce an intended output of some kind. The characteristics of a process are :

- prescribes all major process activities
- uses resources, subject to constraints (schedules), produces intermediate and final products
- may be composed of subprocesses that are linked in some way
- each activity has an entrance and exit criteria
- activities are organized in a sequence
- every process has a set of guiding principles explaining the goal of each activity

- constraints or controls may apply to an activity, resource, or product (examples : budget or time) [PFLEEGER pg. 44 – 47]

Wasserman classifies the fundamental notions in the discipline of software engineering into eight categories as listed below. These notions are referenced throughout the paper.

- Abstraction – description of problem at some level of generalization that allows the software developer to concentrate on the key aspects of the problem without getting mired in the details
- Analysis and Design – provide a means to build models and check them for completeness and consistency
- User Interface Prototyping – building a small version of a system used to help user identify the key requirements of a system
- Software architecture – overall architectural structure of a system. Eases implementation and testing as well as enhances the speed and effectiveness of maintaining and changing a system.
- Software process – different types of software needs difference processes. Various process models are described later in this chapter.
- Reuse - the practice of referencing artifacts used in previous development for the benefit of reducing the development time by utilizing existing requirements, designs, test scripts or data. Also aids in learning from past efforts.
- Measurement – quantifiable description of a satisfactory system

- Tools and integrated environments – software programs used to automate various software engineering activities and technique.

[PFLEEGER98 pg. 29-35]

Another perspective of software engineering is the view of a software project as having five distinct facets by Leszek A. Maciaszek in his "Software Engineering Pentagon" (Figure 2.2). The facets addressed in this paper are the development lifecycle and the modeling languages as they relate to "programming in the medium". The other facets are very important to the development of software project but are outside of the scope of this paper.

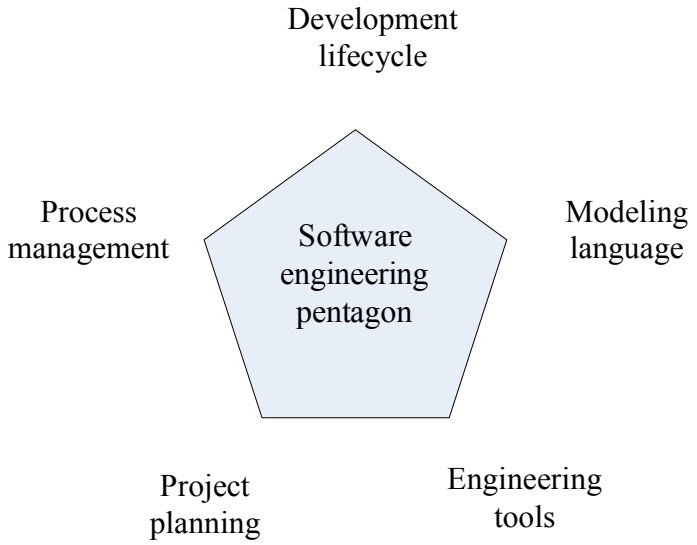

Figure 2.2 Software Development Pentagon [MACIASZEK pg. 3]

2.3 Modeling Languages

"Software engineering is about modeling." [MACIASEK pg. 27] Models are an abstract representation of reality. "By allowing concentration on important aspects of a

problem and by ignoring aspects that are currently not relevant, abstraction allows systematically conquer the problem's complexity." Abstraction is a powerful technique that allows the development team to systematically conquer the complexity of a software project. This technique related to both the process (representation of the software process) and the product (the abstraction representation of the product). This approach applies to software products and processes. The process model defines the lifecycle phases and how they interact, therefore determining what software products needs to be produced by lifecycle phases. A software product model is an abstract representation of a discrete product of a particular lifecycle stage. Examples are requirements, specifications, architectural, detailed design, and the resulting program. Further developing the idea that software engineering is modeling and that the resulting program is actually an executable model, it logically leads to view a programming language as a modeling language. [MACIASZEK pp. 37] This leads to the understanding that the knowledgeable application of modeling languages and techniques is fundamental to software engineering. Process models assist in guiding behavior when working with a group.

Which abstraction techniques to use depend on the paradigm adopted. There are two main paradigms in software development: functional and object-oriented. The functional paradigm uses functional decomposition to break down a complex system into manageable units using the data flow modeling technique. This paradigm also is referred to as procedural, imperative, or structured. In this paradigm, the software model is further divided into decreasing levels of abstraction linked by data flows. The

proven techniques from this paradigm have value in the modern software development world and will be considered through the paper. The object-oriented paradigm breaks the system into components or packages of classes that are linked together by relationships. Abstraction can be modeled with nested structures by allowing components or packages to contain multiple levels of other components or packages. Various object-oriented techniques will be used to illustrate how to reduce a software project into a collection of well-defined, related components in a flexible, maintainable and reusable manner. [MACIASZEK pp.12-13] The modeling languages and notations vary depending on the paradigms the developer chooses, so the choice to follow a functional or object-oriented approach will define the techniques and artifacts used to develop the product.

2.4 Software Development Lifecycle

Software engineering consists of activities which should be present in any development project: Requirements gathering, analysis, design, construction, testing deployment, maintenance. These activities represent phases of the development lifecycle. These phases go by different names or are sometimes consolidated but the activities should occur however labeled. For example, Pfleeger rolls requirements gathering and analysis into one phase called "requirements analysis and definition." Another example would be from Roger Pressman. He compresses the phases into three generic phases : Definition, Development, Support. [PRESSMAN pg. 22] Some experts like Leszek A. Maciaszek contend that "testing … is an all-encompassing activity that applies to all phases of the lifecycle". Therefore testing is not depicted as a

separate phase in Maciaszek's process model in figure 2.3. The following diagram graphically depicts how the phases relate from one major release to another. [MACIASZEK pg. 6] Kulak and Guiney encourage more limited phases. "We consider requirements gathering a separate activity from analysis. This is contrary to several other prominent industry luminaries, who lump them together. Neither way is ultimately correct or incorrect; we have simply chosen to separate these activities to emphasize their importance." [KULAK pg. 5] The IEEE Taxonomy standard provides the greatest number of phases including qualification, manufacturing, installation and checkout, and retirement phases. This level of detail is applicable to some environments. This paper will follow a variant set of activities prescribed by Kulak in order to better delineate the appropriate application of principles and use of techniques. The intent is to show why each phase is important and that following some principles, practices and techniques are essential to generating efficient and qualify software product. Daryl Kulak states the reality of many projects "The emphasis that the team gives to each phase determines the direction and quality of the resulting system. If one activity is not give its due, then will be predictable problems with the project and the end product. In reality, however, certain activities usually receive more attention than other activities. It is not easy to explain why this occurs, but it does. The activities that are usually ignored or paid lip service are: requirements gathering, testing, deployment, maintenance" [KULAK pg. 2]

The following table describes the phases of the software development lifecycle followed by this paper which provides a logical break down of this paper's topics.

Table 2.1 Lifecycle Development Phases

Activity Name	Description
Requirements gathering and analysis	Gather and document the functions that the application should perform in the language and perspective of the user Build a logical solution that satisfies the requirements but does not necessarily take the physical constraints into account.
Design	Begin with the logical solution and change it to work effectively with the physical constraints (network latency, database performance, cashing, availability, and so forth) and produce specifications that can direct the construction effort
Construction	Use the design to produce working code, which involves making the lowest-level design decision, writing code, compiling, debugging, and testing by increment.
Testing	Use the constructed application to produce a complete working system by system testing, detecting and recording issues, fixing problems, and getting user acceptance of the result.
Deployment	Fit the tested application into the production environment by deploying the code libraries to the designed machines, training the users, and fine-tuning the business procedures surrounding the new system.
Maintenance	Administer and make changes to the working system in the production environment to adapt to ongoing business changes (legislative, competitive), technology changes (hardware, software, communications), physical changes (location, configuration), personnel (information technology (IT), user), system issues (code bugs, design problems) and politics.

Maciaszek states that in order to understand the software lifecycle one must understand the context of software production. He captures these fundamental observations in what he refers to as the "quintessence of software engineering".

- The software system is less than the enterprise information system
- The software process is part of the business process
- Software engineering is different than traditional engineering
- Software engineering is more than programming
- Software engineering is about modeling
- The software system is complex.

[MACIASZEK pg 7]

In the following diagram, Maciaszek illustrates how the conclusion of the development of one release of the product leads into the construction of the next release. This diagram is included to visualize how phases relate to one another and to future releases.

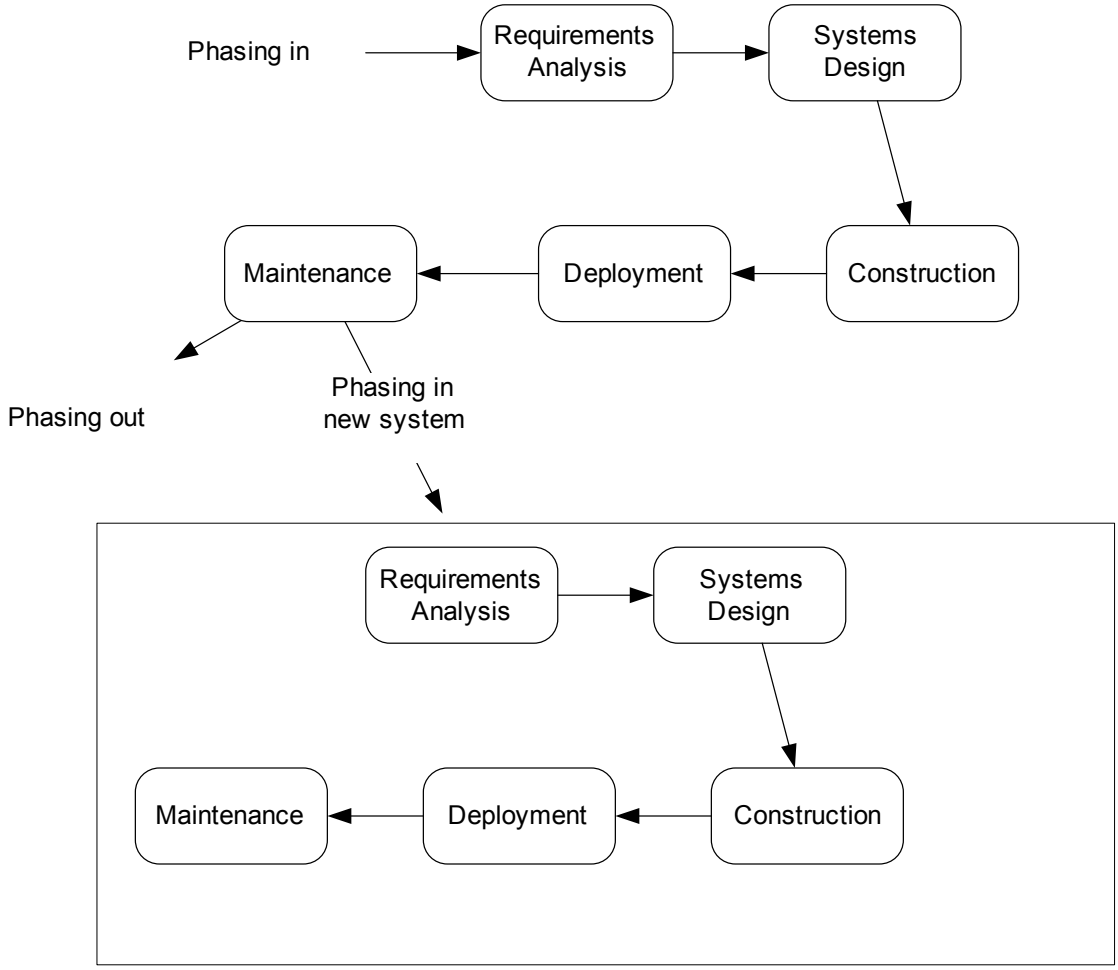

Figure 2.3 Phasing to a New Release [MACIASZEK pg. 6]

2.5 Process Models

In order to communicate the appropriate uses of software engineering techniques and practices, it is helpful to understand the various software process models that are in common use today. Each of these models organizes the activities (lifecycle phases) of software development in different ways, with each serving a particular purpose. Models are used to form a common understanding of each activity, assist team in finding omissions, inconsistencies, and redundancies in the process, thus allowing for

the generation of a more effective model. The process model chosen for a development project should reflect the project goals and should be altered to fit the unique situation of each project. Models are guidelines to help organize a project.

The **waterfall model** was one of the first models introduced [ROYCE70] in 1970 and depicts each of the stages as cascading from each other giving the image of a waterfall. It is derived from the hardware arena which followed a manufacturing perspective. Each phase, or stage, of development is not initiated until the previous phase is completed and approved. The phases include requirements analysis, system design, system design, program design, coding, testing, training and maintenance. Progress is measured in delivered artifacts such as requirements specifications, design documents, test plans, and code reviews. This model has proved to be impractical when implemented for the whole project because software evolves as the problem is understood and alternatives are evaluated. There is the inevitable need to traverse the waterfall and repeat or redo one of the stages. It also imposed management structure on system development. This model can result in significant integration and maintenance issue. Another criticism is that there is no indication how one phase is transitioned to another. However, the waterfall model does provide a good basis for other process models. [PFLEEGER98 pg 48-49]

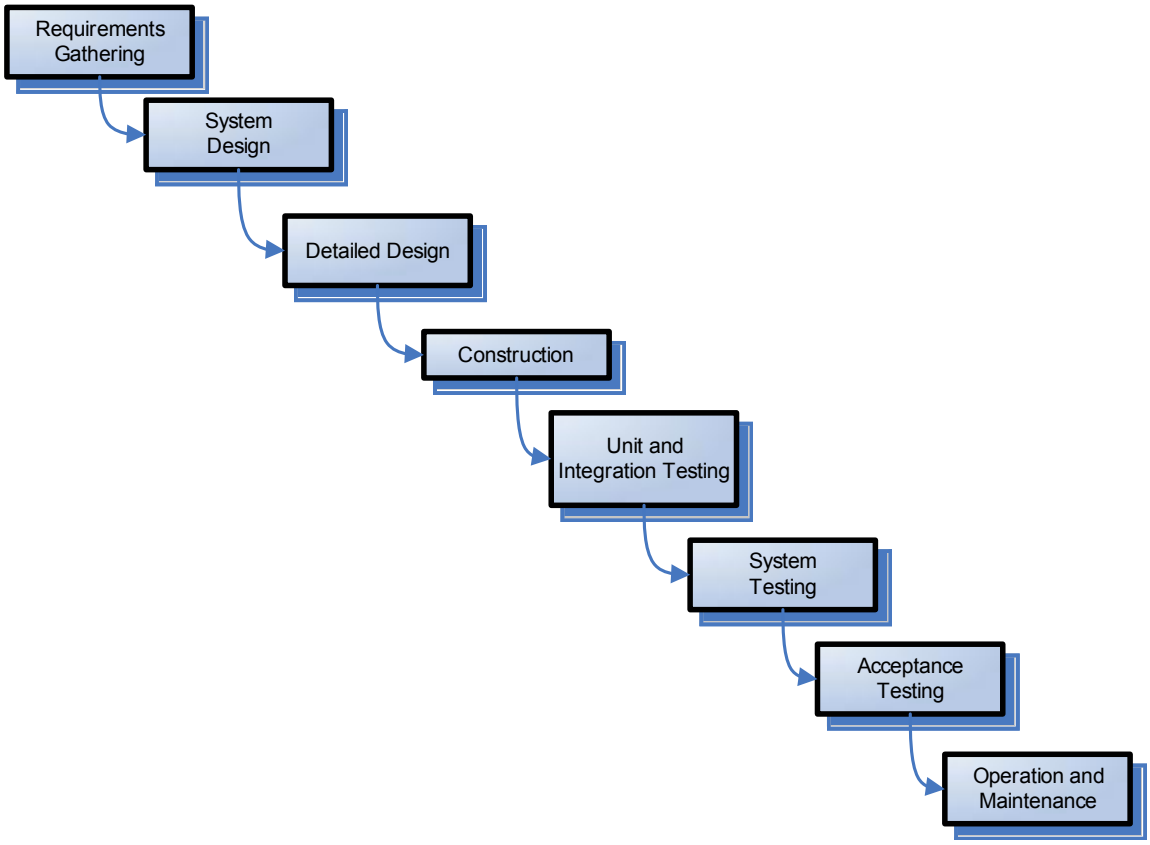

Figure 2.4 Waterfall Process Model

Adding feedback to the waterfall model improves this model by allowing the activities of a previous phase to be revisited based on the feedback of the next phase. For example, if during system design, the developer discovers a new requirement, then he should go back to the requirement "phase", update the requirements and then continue with the system design. The obvious limitation is the further down the cycle a change is recognized, the harder that it is to change in earlier phases.

The **V model** is a variant of the waterfall model which depicts how testing activities relate to analysis and design. The name comes from the shape of the following diagram (Figure 2.5) that illustrates the relationships, arranging the activities

in a V formation. One of the short comings of this model is the fact that requirements are not validated until acceptance testing is engaged. The time interval between acceptance testing and requirements might be to such the extent that any feedback from the user generates very expensive alterations to redesigning the system.

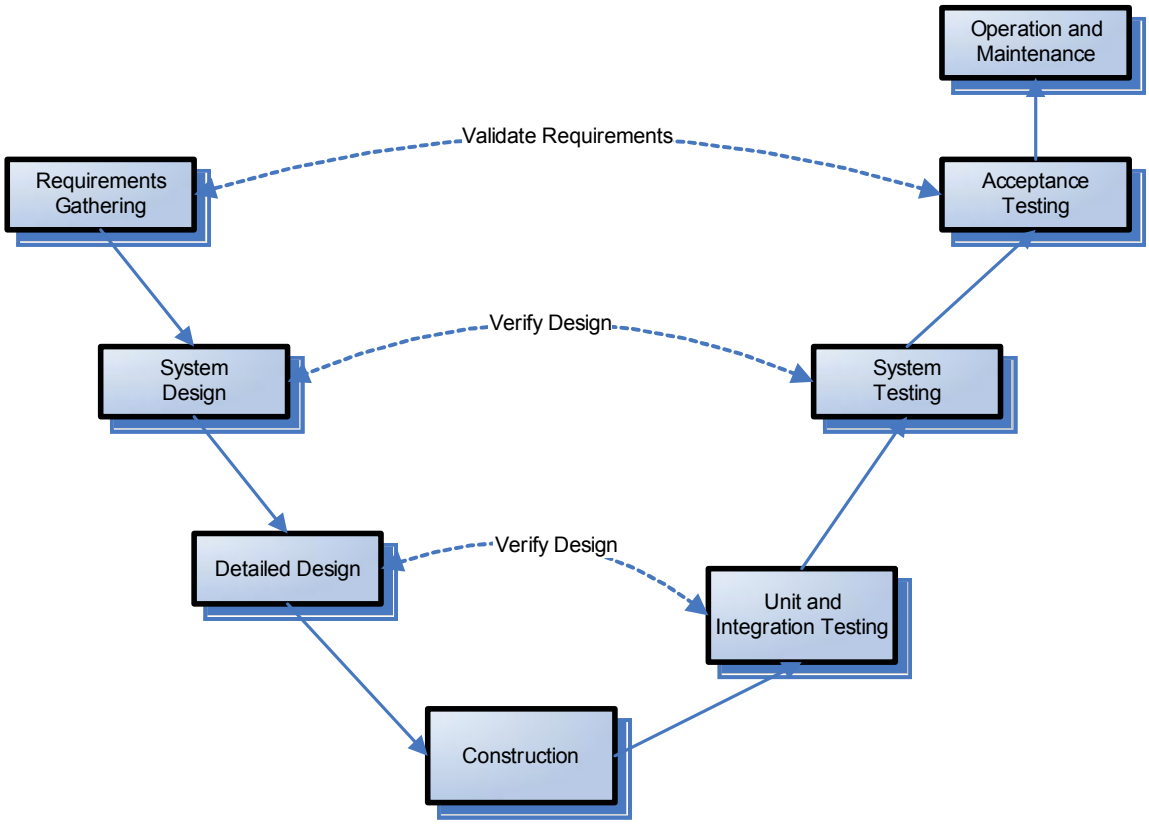

Figure 2.5 The V Process Model [PFLEEGER pg. 52]

The **prototyping model** allows for the quick construction of all or part of a system to understand or clarify the system. It is helpful for reducing risks and uncertainty in development. Each phase is cycled between creating the prototype and the customer creating a list of revisions until both the developer and customer agree.

The criticism of this approach is that it causes the focus to be on the user interface and external functionality rather than the overall functionality of the system and it can be costly to generate a multitude of prototypes.

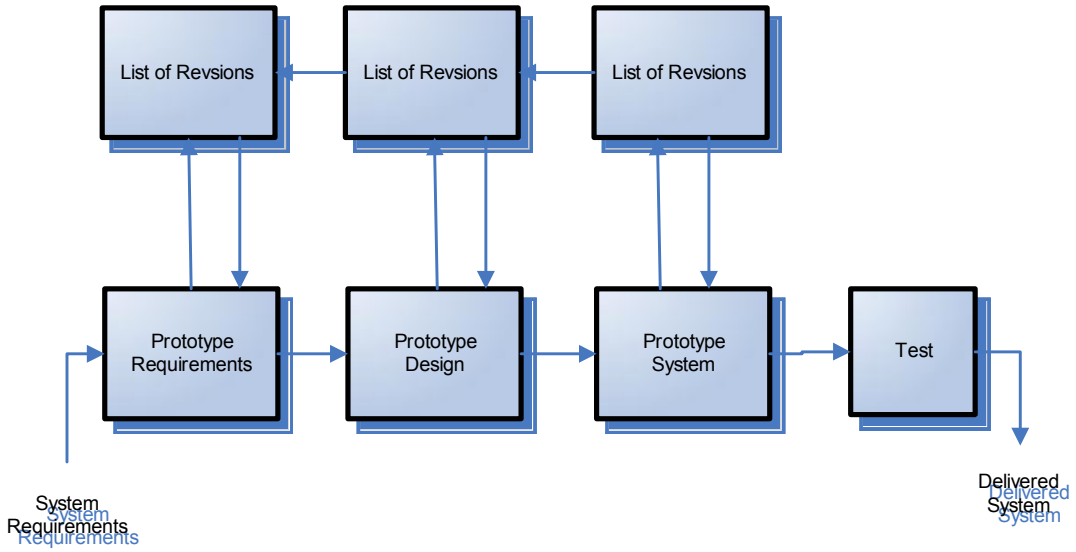

Figure 2.6 The Prototyping Process Model [PFLEEGER pg. 53]

The **phased (incremental or iterative) development model** is useful for satisfying today's customer desire for quick results. It has proved to be very effective approach to modern software engineering. In fact, Fred Brooks reports that nothing as changed his own practice as incremental development. [MCCONNELL2004 pg. 16] Careful planning from the beginning is necessary for this model to define and prioritize the requirements in a manner that can be organized into incremental releases. With this model, there are two systems functioning in parallel: the development system and the production system. Two approaches developers use to organize the requirements and design into incremental releases are the incremental development approach and the

iterative development approach. With the incremental approach, subsets of the functionality are added each release but with iterative approach, new functionality is added to an already complete system. The first release is the simplest possible version of the system. With the iterative model, all functionality is present in each release but the modules are improved with each release. Both have their advantages and ultimately produce the same result but the phased releases will vary in the level of detail of each module.

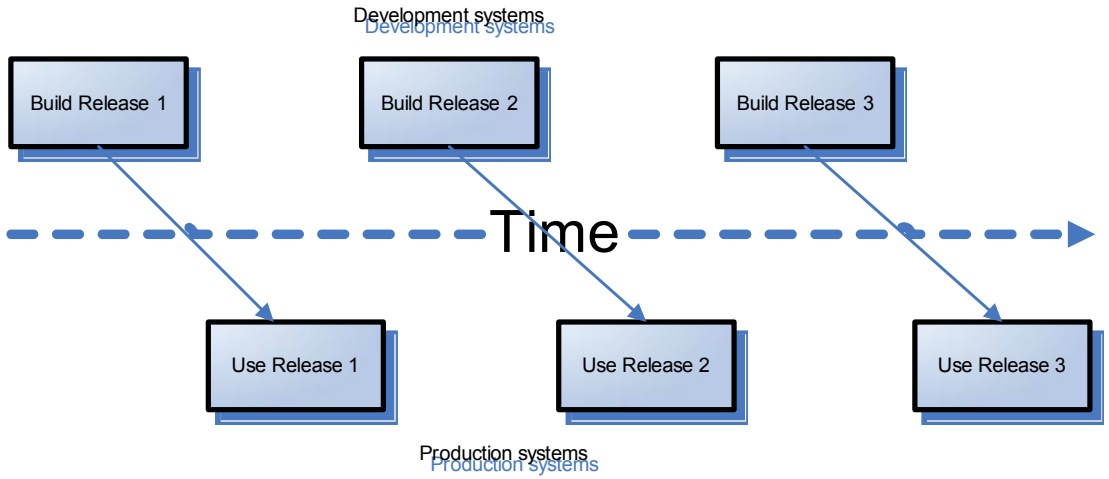

Figure 2.7 Phased Development Process Model [PFLEEGER pg. 56]

The spiral process model is similar to the iterative development approach mentioned above in that it constantly revisits previous phases. The spiral model combines development activities with risk management to control and recognize risk. The start of the project is towards the center cycling through the development activities. As requirements are revisited, another "loop" of the spiral begins, further improving and changing the artifacts of each phase.

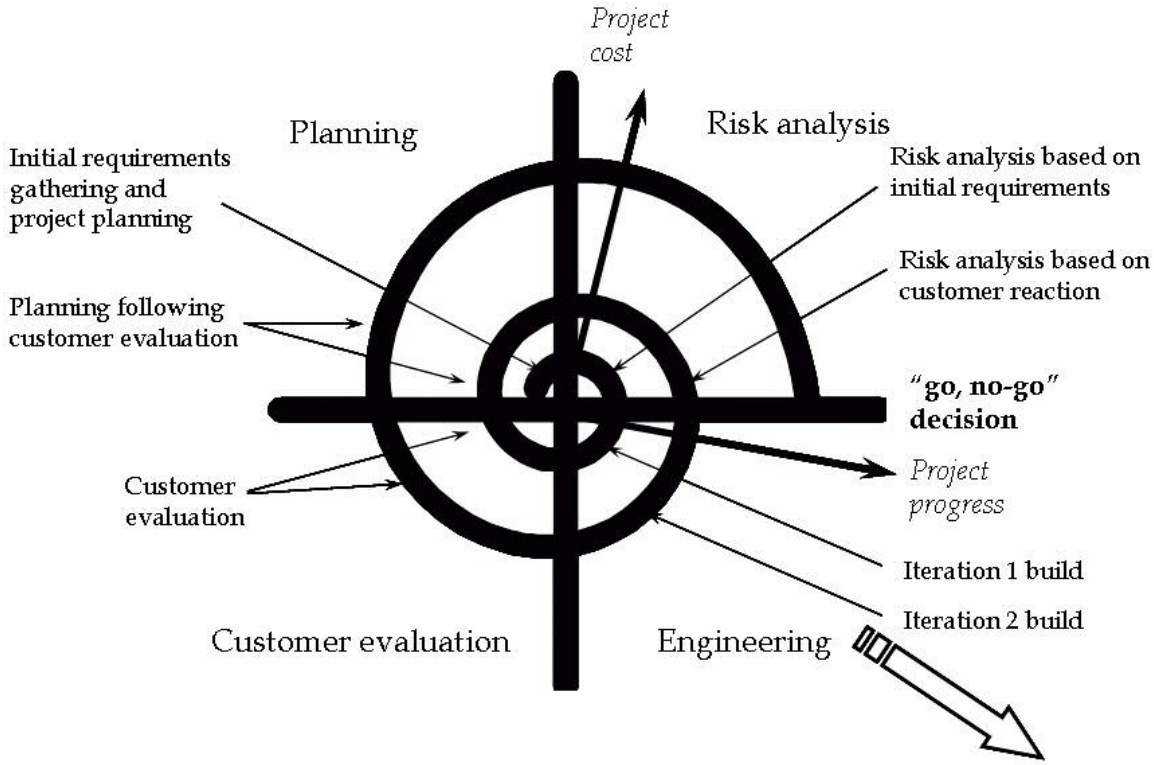

Figure 2.8 Spiral Process Model [MACIASZEK pg. 27]

2.6 The Agile Alternative

In recent years, a conceptual framework for software development called the agile software development as emerged to challenge the tradition process models discussed above. This approach has been applied to several methods and most of these methods emphasize short iterations, each having a duration of one to four weeks. Each iteration is treated as a miniature project and executes all the necessary software development activities required for that iteration. The resulting product from an iteration might not warrant a release but one of the premises of this approach is that the software could be released at the end of any iteration for customer review. The agile approach is to reevaluate the system at the end of each iteration, making the necessary

adjustments to the requirements and design during the next iteration. Documentation is secondary, given preference to face-to-face communication. Teams are usually co-located in a "bull pen" and work together to get the project completed, including the customer, ie. the person that defines the product. The team includes the developers and possibly testers, interaction designers, technical writers and managers. Because the chief metric of progress is working software, little documentation is generated which is the main criticism of this approach. The founders of this process created "Manifesto for Agile Software Development" in 2001, which is listed below.

> Manifesto for Agile Software Development
>
> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools
> - Working software over comprehensive documentation
> - Customer collaboration over contract negotiation
> - Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.
>
> Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt,

Agile methods are often criticized as undisciplined and at the opposite end of the development spectrum from the "plan-driven" methodologies. It is more accurate to say that agile methods are on the "adaptive" end of the spectrum rather than on the "predictive" end, with the focus being on the changing realities presented as the project evolves. In contrast, predictive methods require great effort to introduce change since they are based on planning for the future. While agile methods have a similar flow to the iterative development process, the agile approach focuses on release cycles in weeks, often with a fixed time increment (a strict time box), ie. the team will always have a release ready every two weeks. The emphasis is on creating a basic working version of the product and continually improving it. Some teams that follow the agile approach implement the waterfall model each iteration, going through the complete lifecycle in the small time box. Other variations work on various development activities simultaneously, not adhering to any of the traditional process models. Agile teams may appear "undisciplined" but often do follow very disciplined and defined processes therefore distinguishing it from "cowboy coding" or hacker status. In fact, each iterative release is carefully planned for customer evaluation. A major delivery

resulting in the product being put into production has a duration of about six two-week cycles.

Agile development works best when the developers of a small team (less than 10) are physically co-located and the project has fuzzy or unpredictable requirements that can quickly change. A team should consider the agile development approach if the team is composed of a small number of developers, the developers are all experienced, the project has low criticality, the expected frequency of requirement changes is high and the team's culture thrives on chaos.

2.6.1 Extreme Programming Method

Here are a few examples of methods that have embraced the agile development approach: Extreme Programming, Scrum, Agile Modeling, Lean software development. [WIKIPEDIA AGILE] At the time of this writing (circa 2006), Extreme Programming (XP) has received sufficient attention to warrant reviewing its implementation of the agile development approach. Kent Beck, one of the founders of the Agile movement, describes XP in his book "Extreme Programming Explained" [EXTREME pp. 37-70]. He defines XP as a lightweight methodology for any size teams developing software in the face of vague or rapidly changing requirements. The core practices of XP include those of agile development as well as simple design, pair programming, ten-minute build, test driven development, design improvement, collective code ownership, coding standards, and working at a sustainable pace. Pair programming and collective code ownership are two of the more radical ideas of XP. Pair programming means, according to Kent, "write all production programs with two

people sitting at one machine. … Pair programming is a dialog between two people simultaneously programming (and analyzing and designing and testing) and trying to program better." [EXTREME pg. 42] Most programmers can only do pair programming for more than five hours a day. Pairs should rotate frequently, every one to two hours at natural breaks in development. The benefits of this practice are that it keeps each other on track, allows for brainstorming to refine a system, clarifies ideas, holds each other accountable and allows one to take initiative when the other is stuck. Sharing time in this manner also gives the group ownership of the code, creating built-in redundancy. The criticism of this practice is duplication of effort and inefficient use of resources.

Another practice promoted by XP is the ten-minute build. The product should be able to be built and tested in ten minutes. This practice requires automation of builds and tests. It is not always easy to get the build and test cycle down to ten minutes but should be given priority to find a way to accomplish this. If it takes much longer than ten minutes, then it is more likely to be used less often, resulting in longer intervals between builds.

The XP practice of test-first programming addresses several issues. Writing code to test for each condition of a method reduces scope creep, promotes loosely coupled, highly cohesive code, builds trust because the code is proven to work, and establishes rhythm (test, code, refactor, test, code refactor, …).This practice coupled with the ten-minute build will greatly assist in preventing unexpected behavior to be introduced by other modules. While a pure XP approach is not be appropriate for every

team, this method should be reviewed and some of the applicable practices can be adopted to improve the efficiency of any team.

Each developer has his own style. Each development team has its own dynamics. Each development project has its own unique set of circumstances. So it stands to reason that no one single approach to software development fits all situations. The small development team might be tempted to not follow any process model because of the difficulty of adopting the discipline required. The team members should realize that improvement can come from following software engineering and software engineering requires some level of discipline. Therefore it is important that any development team that desires to improve the quality, stability and efficient of its products to adapt to today's software engineering principles and practices.

2.6.2 Task Swapping: Alternative to Pair Programming

Task swapping is a concept that might be worth considering for the small development team. Developers with similar skill levels are paired together. Each pair is responsible for the development of at least two projects or components. Each developer would work isolated for no more than 3 hours and then send 30 minutes or so of transition time to review and discuss alterations to the projects, then the developers will swap project. This practice would be a variant of pair programming in which "lone developers" trade projects with minimum overlap. This approach might satisfy management's concern with the "wasted resources" perception of paired programming and have the advantage of reducing code ownership. It has the build-it emphasis on

documentation and following coding practices since the two programmers would have to communicate on each other's progress.

CHAPTER 3

UNIVERSAL PRINCIPLES AND PRACTICES

Principles are what define and drive quality software. They evolve as the discipline matures. Old ones are modified or discarded as new ones are added. Some principles common in the 1960s would not be acceptable in today's software engineering word, such as keep variable names short, or do whatever possible to reduce the length of a program. [DAVIS95 p. 5] No doubt, some of today's principles held as paramount will be discarded as the discipline continues to grow. While it may be difficult to do, software engineers of a small development team would be advised to stay abreast of the current principles in order to continue to product quality products in an efficient manner. Some of the software engineering principles are "universal" and apply to all or most phases of the development lifecycle as well as to any size development team. Other principles are specific to the respective phase of software development mentioned in the previous chapter. Since the focus of this paper is on the technical aspects of software engineering, the management and product assurance principles will not be described. This chapter reviews the universal, or general, principles of software engineering that can apply to the technical aspects of several or all of the software development phases. The principles that fall into one of the development lifecycle phases will be discussed in its appropriate chapter.

As defined in the first chapter, principles are truths, rules or assumptions that hold true regardless of how the software is implement. If a principle is violated, then it is expected that some aspect of software engineering is compromised, resulting in an inferior product or significant deadline overrun. It is important that the small development team be cognizant of these principles, which provide the underlying guidelines for decisions and effectiveness. In a small company, the violation of these principles can be devastating, resulting in a poor reputation or financial ruin.

Quality of product must not be compromised. According to Alan Davis, there is one overriding principle that governs all other principles of the software engineering discipline: a product must meet the quality specified in the requirements or it is a failure. The reason to prescribe to any principle is to efficiently produce a product that meets the acceptable quality. However, the definition of quality is "in the eyes of the beholder". [DAVIS95 pg. 9]. Developers might define quality as elegant code or design whereas customers would see quality in light of how a product satisfies his/her needs (real or perceived). Therefore though difficult, some definition of quality must be understood by all team members, management and customers in order to product an acceptable product. To honor this principle, acceptable quality must not be sacrificed to achieve a higher level of productivity. It is interesting that Alan Davis goes on to say "As the attempts are mode to drive productivity up, the density of bugs increases." [DAVIS95 pg. 10] One of the objectives of this paper is in direct conflict with this statement. The underlying assumption of this paper is that by applying modern principles and techniques to any software project, the productivity will increase

as well as the quality of the resulting program. In fact, it is assumed that only by following best practices and principles, that a software development team can sustain long term quality products that are efficiently produced. Quality should be a driving factor from the initial phases of the development, not an after thought. An example of this principle is that a prototype used for gathering or defining requirements should not be evolved into a product. While efficiency is to be considered at all phases, the reliability of the product should always come first. Therefore if a practice or technique would provide a more efficient way to achieve a desired result but would compromise the reliability of the product, then it should not be utilized.

Obtain customer feedback early in the development lifecycle. The earlier that a product can be reviewed by customers, the sooner the development team will receive feedback, and therefore have opportunity for the modification to be incorporated into the product in a cost-effective and efficient manner. The major flaw of the waterfall model is that the customer does not see the product until the end of the lifecycle cycle, therefore requiring much more effort to apply any feedback from the customer, thus violating this principle.

Don't use the first release. When creating an entirely new product, expect the first release not to be acceptable, therefore should be treated as a prototype. This principle requires that the development team allow time for the first cut not to be the final cut. The principle of building software incrementally has been popularized in recent years with the advent of Extreme Programming [EXTREME]. However this concept is not new. In 1971, R. Mills suggested a similar approach: Start small, with a

working system that implements only a few functions and incrementally add larger subsets until the final product is arrived. [MILLS] Following this practice effects all lifecycle phases. For example, all documentation should be stored in a manner that can be easily retrieved and changed since requirements changes are inevitable, and requirements must be changed before design can be altered.

Design should be intuitive. Well designed software should be intuitive to use. One measure if this principle is being followed is the size of the user manual: the smaller the manual, then more intuitive the product. This principle is universal because it spans more than just the design phase of the project, but also effects how requirements are organized and how testing is accomplished.

Document assumptions. Each component should have an explanation of the assumptions of a product or its environment Manny Lehman states that developers "make one assumption every 10 lines of code". [LEHMAN91 pg. 243-258] Therefore, it is impossible to be cognizant of all the assumptions that are made at each phase of the project. However the practice of maintaining a diary of the recognized assumptions and the resulting implications allows the developer to quickly identify the impact of a faulty assumption. A good practice would be to include an impact assumption list with each review and to provide an assumption section in each design module.

Use appropriate techniques for the project. One advantage of a small development team should be its ability to be flexibly and change techniques and languages to adapt to the situation. The desire to find a simple solution to a complex problem can lead a developer to attempt to use the same notations for software

presentation throughout the entirety of a development cycle. It would better serve the development team if its members were aware of several techniques and practices for each phase of the development process and establish a principle that one set of techniques and languages is to be selected for each phase of a project. Certainly if a notation is optimal for more than one phase, it should be used where applicable. But a team should not dictate that one set of techniques and languages should be required through every phase of every project. This practice has received some criticism however with the emphasis on incremental development. The argument is that instead of throwing the product away, make the necessary adjustment for the next cycle of development and release an improved product. The approach would be based on the amount of change requirement and the type of software being generated. Business system do not require as formal execution of construction prerequisite actvities as do mission-critical system or embedded life-critical systems.

Manually prove a technique before automating. Before adopting a new tool, the technique that the tools is implementing should be applied by hand first, to demonstrate its usefulness and convince management that this technique is worthy to automation. It is tempting to purchase a flashy tool before proving the technique is useful with the team's environment.

Avoid gold plating. A principle particularly applicable to small development teams is to stop when the goal is achieved. The addition of unspecified functionality or quality is called "gold plating" and while it might provide a temporary sense of satisfaction of "over delivering" a product, it has the potential of introducing additional

bugs or complexity that the customer neither desires or appreciates and it will inevitably add to the cost of the project and possibly introduce unnecessary bugs.

Use industry standard document formats. Using software engineering standard documents can be very helpful if for no other reason that serving as a check list to avoid major omissions. Standards are discussed later in this paper, reviewing some of the industry standards and suggesting what is applicable to the small development team.

Reuse. Reusing software components from previous products is an obvious practice that can result is reducing the time required and improving the quality for all phases of a project. This practice is not limited to the confines of the construction phase but dictates how requirements and design activities are organized and effects testing. Once a small software development team adopts reuse as a team norm, then stands to reason that reuse can greatly improve a team's ability to produce efficient products by not only learning from past products but borrowing from them as well. Later chapters will provide more discussion of reuse.

Fix defects as early as possible. Steve McConnell backs up this principle with the following statement : "Researchers … have found that purging an error by the beginning of construction allows rework to be down 10 to 100 times less expensively than when it's done in the last part of the process, during system test or after release." [MCCONNELL pg. 29] The following table shows the relative expense of fixing defects at various phases of the project.

Table 3.1 Average Cost of Fixing Defects Based on When They're Introduced and Detected [MCCONNELL2004 pg. 29]

	Time Detected				
Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	-	-	1	1-	10-25

CHAPTER 4

SOFTWARE ENGINEERING STANDARDS

Software engineering experts have researched and developed practices and techniques using various notations and instruments to aide in the exercise of each phase, resulting in software standards. Practices and techniques reflected in the standards evolve from the application of good engineering principles to software development. Principles of engineering were used to build principles of software engineering which led to practice standards. The detailed implementation of the provisions of the practice standards results in the creation of best practices. [MOORE pg. 5].

A standard is defined as 1) an object or measure of comparison that defines or represents the magnitude of a unit, 2) a characterization that establishes allowable tolerances or constraints for category of items and 3) a degree or level of required excellence or attainment. Standards are definitional in nature, established either to further understanding and interaction, or to acknowledge observed norms of exhibited characteristics or behavior. [MAGEE2006]

Software engineering standards are used to improve communications between software engineers and others, achieve economy of cost, human effort and essential materials, institutionalize practical solutions to recurring problems, achieve predictability of cost and quality, and to establish norms of acceptable professional practice. [IEEE1002 foreword] The purpose of a standard is to communicate the

collective knowledge of industry experts on a particular subject, not to lock a developer into a single way of accomplishing a task. Each standard has to be evaluated and adapted to fit the needs of the team. The small development team can benefit from adopting some standards by using the contents to reveal omissions or activities out of sequence, by providing a common format to record information at each phase, by having documents that are linked to together by a uniform approach to development. The proper use of standards can improve and evaluate software competency by increasing quality, customer satisfaction, reducing cycle time and increasing productivity. Standards provide a common framework and terminology for the development team and customer to more efficiently communicate.

However, choosing the appropriate standards for a small development team to implement can be difficult. At the time of this writing (circa 2006), there are over 350 potentially applicable standards in the software engineering realm with over 50 organizations responsible for monitoring and maintaining these standards. "Many organizations become 'lost' trying to select the right set of standards to use to produce quality software products at a reasonable cost." [MAGEE1997] With such a large variety standards, it is practical to review and choose between a few of the most popular ones or ones that are already familiar to the team members. Other option is to employ experts like Software Engineering Process Technology [SPET] to review a company's needs and recommend a set of standards to follow. However obtained, adopting some level of industry standards is an excellent way to improve efficiency by learning from

the collective wisdom of software engineers. The level of implementation of standards can only be determined by the team itself.

The lists below are documents describing standards related to software engineering. Other standard organizations exist that are helpful to the small development team but only these two organizations are mentioned below will be mentioned in this paper for brevity.

4.1 International Organization for Standardization (ISO)

ISO is a non-governmental organization with representation from 156 counties which acts a bridge in which a consensus can be reached on solutions that meet both the requirements of business and the broader needs of society, such as the needs of stakeholder groups like consumers and users. ISO prides itself on being voluntary, market-driven, equal footing, consensus based, and world wide. Often its standards become law. One of the ISO areas of standardization is technology, which involves software engineering. Below is a list of standards documents produced by ISO that influence the software engineering community. The ISO documents are listed for reference but not reviewed in this paper. [ISO]

- ISO/IEC 2382-1 Vocabulary – Fundamental terms
- ISO/IEC 2382-7 Vocabulary – Computer Programming
- ISO/IEC 12207-1995 - Software life cycle processes
- WD 14764 - Software Maintenance
- WD 15288 – Software Life Cycles

4.2 Institute of Electrical and Electronics Engineers, Inc. (IEEE)

IEEE has developed and maintained many standards for the software engineering disciple since the early 1950s. "The existence of an IEEE Standard does not imply that there are not other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard." [IEEE830 pg 2] However, these standards are rigidly followed by large organizations and therefore are worthy of review to evaluate what qualities of the reported tools used might apply to the small software development team.

The following IEEEE standard documents were selected for review as each provides a tool that is directly applicable to some phase of the software development cycle.

- Standard Taxonomy for Software Engineering Standards [IEEE1002] – explains how IEEE classifies software engineering standards into the following partitions : process standards, product standards, professional standards, notational standards. This document divides software engineering into job functions and software life cycle.

- Standard glossary for Software Engineering Terminology [IEEE610] – identifies the terms used in the computer field and to establish standard definitions for these terms.

- Standard for Software Verifications and Validation Plans [IEEE1012] – defines how to perform verification and validation plans for comprehensive evaluation throughout each phase of a software project.

- Recommended Practice for Software Design Descriptions [IEEE1016] – describes the necessary information content and recommendations for an organization of Software Design Descriptions (SDD)

- Standard for Software Reviews [IEEE1028] - defines five types of software reviews, together with procedures required for each type. The five types are : management reviews, technical reviews, inspections, walk-throughs and audits

- Software Requirements Specification [IEEE830] – describes alternative approaches to the specification of software requirements in order to help software customers to accurately describe the product, to assist software suppliers to understand what the customer wants, and to guide individuals on the development of a software requirements specification (SRS) for their own organization as well as other supporting items like a SRS quality checklist and SRS writer's handbook.

- Standard for Software User Documentation [IEEE1063] – provides the minimum requirements for the structure, information content, and format of user documentation, including both printed and electronic documents used in the work environment by users of systems containing software.

- Standard for Developing Software Life Cycle Processes [IEEE1074] provides a process for the creation of a software life cycle model including the creation of the software life cycle, primarily directed at the process architect for a given software project.

- Standard for Software Maintenance [IEEE1219] – describes the process of managing and executing software maintenance activities.

- Standard for Software Test Documentation [IEEE829] – provides guidance on the development of test plans. Describes a set of basic test documents and specifies the form and content of individual test documents.
- Guide to Software Reuse [IEEE1420] – defines the minimal set of information about assets that reuse libraries should be able to exchange to support interoperability.

Each member of the small development team should review the appropriate standards for his or her role to better understand the scope of responsibilities within each role. The standards documents provide a concise description of the industry's perspective of that role. While some of these standards are dated and have been replaced with more modern practices, the majority of the information remains applicable to current software engineering development.

CHAPTER 5

CASE STUDY INTRODUCTIONS

To better understand how the software engineering principles and techniques addressed in this paper can be applied to the small development project, several case studies have been reviewed and highlighted as to how software engineering practices effected the projects. Theses case studies are referenced in the remaining chapter.

5.1 Movies on the Web

This case study was derived from the example used in Maciaszek and Liong's "Practical Software Engineering: A Case Study Approach" book. [MACIASZEK pg. 38] This paper elaborates on this case study to develop examples of various techniques. This example is referenced the most of the case studies and provides a running example throughout the remaining chapters.

This system is used by cinema chains to advertise movies on the Internet, to provide movie information and screenings so that customers can query movie info and order tickets. Most of the object orient diagrams in the General Software Engineering Techniques chapter are drawn from this example.

5.2 Selling of Advertising Time for Piccadilly Television

This case study is taken Shari Lawrence Pfleeger's summary [PFLEEGER pg. 35-37]. The original case study was documenting in the book entitled "Complete Systems Analysis: The Workbook, the Textbook, the Answers" by James and Suzanne Robertson (1994). [ROBERTSON] It serves as an information system example and involves the selling of television airtime for a regional British television company.

Piccadilly Television was an 8-year franchise to a commercial television company with rights to a region in the middle of England. The franchise broadcasts a set number of hours of drama, comedy, sports, children's and other programs. The commercial advertisers in the area have several choices to reach the Midlands audience – Piccadilly, cable or satellite. One way to attract commercial advertisers is to publish audience rating (number of types of viewers at different times of day). Ratings reported in terms of program type, audience type, time of day, television company, etc. Bulk purchase lowers rate per hour also effect the advertising rate. Other limitations are ads for alcohol only after 9 p.m., actor in ad cannot be shown within 45 minutes of same actor in show, and only one commercial of a given class may been shown during a commercial break.

5.3 Center/TRACON Automation System (CTAS) [JACKSON]

This case study was written as a summary of an MIT class project to see how application of modern software engineering techniques would benefit a large system. CTAS is a suite of tools to assist air traffic controllers to manage air flow at large airports. The inputs for the system are location, velocity, flight plans, weather data,

runaway info, controller commands. It models the descent rates to predict aircraft trajectories up to 40 minutes in advance, suggesting landing sequence to minimize unused landing slots. It was implemented at DFW airport, improving the landing rate by 10%. This system was composed of two tools. The low-altitude controllers manage air space near airport use the Final Approach Spacing Tool (FAST) and the high-altitude controllers use Traffic Mgmt Advisor (TMA). Both components used same set of software components which are:

- Communications Manager (CM) – message switch moving data among other components, maintains aircraft info,
- Input Source Mgr (ISM) collates input data streams (radar feeds/flight plans) are the Route Analyzer (RA) and Trajectory Synthesizer (TS) – predict paths and arrival times
- Dynamic Planner (DP) – computes runway assignments and suggested delays

CM is main process. FAST and TMA are dependent on CM. The case study focused on the CM component and emphasized redesign which is discussed in design phase and construction phase chapters.

CHAPTER 6

GENERAL TECHNIQUES

This chapter describes many popular techniques used to develop software that are not limited to the scope of one phase of the development lifecycle. Many of these techniques are associated with artifacts which are in included as an example. Structured techniques will be discussed first, followed by object-oriented techniques Since it has already been established that software engineering is about modeling, it is appropriate to classify the techniques as modeling techniques. There are two classifications of models: static and dynamic. Static models depict the process, illustrating that inputs are transformed into outputs. Dynamic models involve the process so the user can visualize how intermediate and final products are transformed over time.

6.1 Software Reuse

Reusing software components from previous products is an obvious practice that can result is reducing the time interval all phases of a project. While this practice has received increased attention in recent years, it is not a new practice. Back in the mid-1950s, a user organization for scientific application of IBM "mainframe" was formed with one of its primary functions being a clearinghouse for contributed software subroutine. This system became one of the first libraries of reusable software. [GLASS pp. 43-44] Reuse-in-the-large is much more difficult, but developing smaller, more generic components is manageable.

Consistent reuse of not only classes but of requirements lists, design diagrams and test plans can lead to reductions in software development code and cycle time and can promote software quality. Throughout the research for this paper, reuse is listed as one of the chief means of improving a software development team. One study found that small development teams of four to eight developers obtained substantial reuse with a strong central architect as the reuse guardian. [FICHMAN] There are several reasons why software is not reused.

1) Easier to create than find. If a development team does not have standards in place on how to organize components, then it may very well be faster for a programmer to recreate a module instead of finding.

2) Behavior of the existing module uncertain. If the behavior of an existing module is not documented sufficiently, then it might be more prudent to recreate the module so that the results are guaranteed. In order to efficiently reuse a module, all assumptions and behavior must be specified clearly.

3) Takes longer to make code reusable. In order to allow other developers, or even the originator, to reuse a module, the module should adhere to the team's conventions and provide the necessary documentation that will provide efficient and effective future use of the model. Robert Glass states that "it is three times as difficult to build reusable components as single use components and that a reusable component should be tried out by three different applications before it will be sufficiently general to accept into a reuse library. [GLASS pg. 49]

4) Uncertain of responsibility. Developers might resist using existing modules because of the liability of unintentionally misusing the module.

5) Difficulty of sufficient documentation. This reason is echoed as a theme through the previous reasons. Developers might be under pressure to just get the code done and not take the time to document the module in a manner in which others would have confidence to reuse it. [PFLEEGER98 pg. 29-35]

6) Conflict with top priorities. Top priorities on projects are to get them released on time and within budget. Reuse is often viewed as ”nice to have” but should slow down the project to get it started.

7) Reuse across teams is hard to coordinate and ownership issues cause disincentives.

6.2 Lai Notation

Lai notation provides a static technique for modeling any process at any level. It is based on the idea that people perform roles and resources provide activities, resulting in the production of artifacts and provides notation to depict the relationships between these entities. State tables reveal the status of each artifact. Lai defined seven types of process elements: [LAI]

1. activity – something will happen
2. sequence – it will happened in some order
3. process model – a view of a particular interest or perspective
4. resource – an item, person or tool related to the system
5. control – some external influence that alters or initiates behavior

6. policy – high level constraints dictating system behavior

7. organization – hierarchical mapping of physical entities to logical entities.

Techniques used in this notation are the artifact definition form and the transition diagram. Using this technique, all possible combinations of values are explicitly stated and noted in the table. One of of this technique is to evaluate the risk of a requirement. In the template below, each sub-artifact will only be assigned two states for simplicity,: low and high for probability, and small and large for severity.

Table 6.1 Artifact Definition Form for Artifact "Risk" [PFLEEGER pg. 70]

Name	Risk (Problem X)	
Synopsis	This artifact represents the risk that problem X will occur and have a negative affect on some aspect of the development process.	
Complexity Type	Composite	
Data type	(risk_s, user_defined)	
Artifact-state list		
Low	*((state_of(probability_x) = low) ((state_of(severity_x) = small)*	*Probability of problem is low, severity problem impact is small.*
high-medium	*((state_of(probability_x) = low) ((state_of(severity_x) = large)*	*Probability of problem is low, severity problem impact is large.*
low-medium	*((state_of(probability_x) = high) ((state_of(severity_x) = small)*	*Probability of problem is high, severity problem impact is small.*
High	*((state_of(probability_x) = high) ((state_of(severity_x) = large)*	*Probability of problem is high, severity problem impact is large.*
Subartifact list		
	Probability_x	*The probability that problem X will occur.*
	Probability_y	*The probability that problem Y will occur*

This form communicates what conditions determine the level of risk. This table assists in determining whether a requirement is worth the risk. Other aspects of the development process can be defined and use diagrams to illustrate the activities and interconnections. Using the combination of the spiral model and risk table can be used to evaluate risks periodically.

The following artifact definition form defines the possible relationships between the movie, actor and listed_as objects in the Movie Actor example.

Table 6.2 Lai Notation – Artifact Definition Form – Movie Actor Example

Name	Create Movie Actor	
Synopsis	Represents the process that creates a Movie Actor	
Complexity	Composite	
Data type	(movie.movie_code, number) (actor.actor_code, number) (listed_as.actor_code, number) (listed_as.movie_code, number)	
Artifact-state list		
Optimal	*((state_of(movie.movie_code) = non null and unique) ((state_of(actor.actor_code) = non null and unique) ((state_of(listed_as.movie_code) = matching value in movie.movie_code) ((state_of(listed_as.actor_code) = matching value in actor.actor_code)*	*Movies are connected with actors via the listed_as entity*
Permitted but not useful	*((state_of(movie.movie_code) = non null and unique) ((state_of(actor.actor_code) = non null and unique) (((state_of(movie.movie_code) = no matching value in listed_as.movie_code) OR ((state_of(actor.actor_code) = no matching value in listed_as.actor_code))*	*A movie does not have any actors. OR an actor is not assigned to any movies.*

Table 6.2 Continued

Corrupt	*(((state_of(listed_as.movie_code) = no matching value in movie.movie_code) OR ((state_of(listed_as.actor_code) = no matching value in actor.actor_code))*	*An unknown actor is assigned to a movie OR An unknown movie is assigned to an actor*
Subartifact list		
	Movie	*Stores movie attributes*
	Actor	*Store actor attributes*
	Listed_as	*Stores the relationship between movie and actors*
Relations list		
Listed_as Movie	*Listing of movies and their assigned actors*	
Listed_as Actor	*Listing of actors and their assigned movies*	

The following transition diagram is used to visually show the conditions that transfer from and to an optimal state for the Movie Actor example.

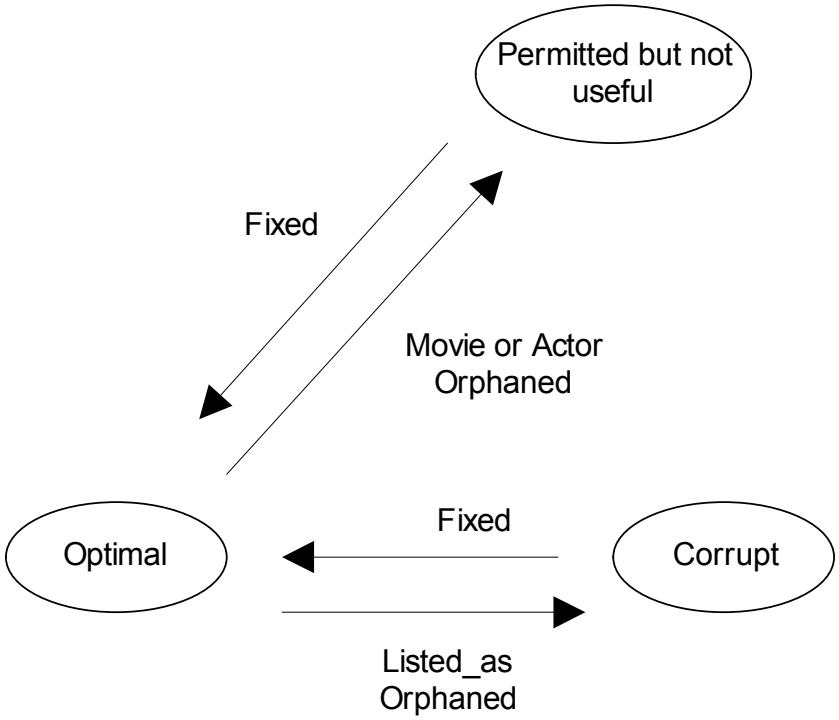

Figure 6.1 Lai Notation – Transition Diagram – Movie Actor Example

6.3 Structured Notation

The traditional structured approach has several notations that have proved useful for several years. Three diagrams used in conjunction are helpful to functionally decompose the problem to arrive at a design for the solution.

6.3.1 Data Flow Diagram

The *data flow diagram* (DFD) has been one of the most popular modeling techniques used in software engineering. It is being used less frequently with the advent of the UML but still remains a valid technique. DFDs are based on functional decomposition which is the method of gradually defining the functional processes of a system. It is a top down activity, starting with a context diagram and culminates with module specifications, therefore combining analysis and design. While DFD

concentrate on processes, it uses data flows thoroughly define the data structure, resulting in an almost complete description of the system. To complete the description, *structure charts* are used to define the intersection of design and implementation and the entity-relation modeling is diagrammed with entity-relation diagrams

The *context diagram* represents only the process that corresponds to the system being developed. It helps determine the place of a system within the environment by showing the inputs and outputs with external entities, like organizations, departments, people, other systems. These diagrams have four elements: data flows (depicted with arrows), external entities (depicted with boxes), processes (depicted with circles), and the data store (depicted with a box with side rectangles). The following context diagram shows the basic activities of the Movie Actor example:

- a customer requesting details on a specified movie
- a customer purchasing a ticket for a specific screening of a movie
- distributors updating the movie database

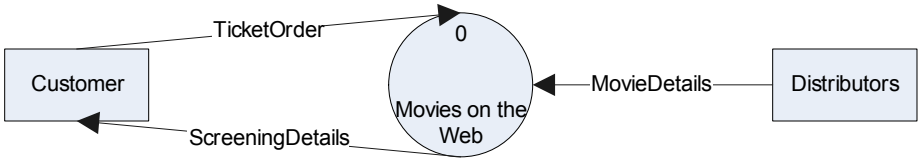

Figure 6.2 Context Diagram – Movie Actor example

All the processing is done inside the system depicted as a circle. The system is expanded in a level 0 diagram (or overview diagram) which provides the specification of what activities are performed inside the system. The Movie Actor system has three processes: CRUD Move Actor, CRUD Screening, Manage Ticketing. These processes

are numbered to uniquely identify each but do not imply any sequence. Processes of the level 0 diagram are each expanded into a level 1 diagram, showing the process decomposed into more defined processes.

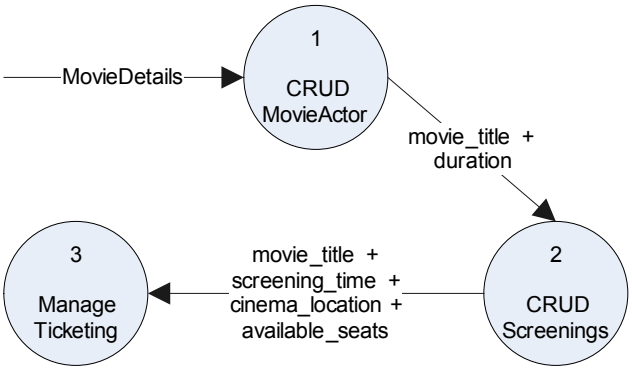

Figure 6.3 Overview (or Level 0) Diagram – Movie Actor Example

Each level 0 diagram can be further decomposed into level 1 diagrams. Keeping with the Movie Actor case example, the CRUD Movie Actor process is diagramed in Figure 6.3 introducing the data store element, in this case the Movie Actor database. There are five (5) processes in this diagram, symbolized by the process circle. The process 1.4 Retrieve Movie Actor provides the movie_title and duration required by the Process 2 of Figure 6.3.

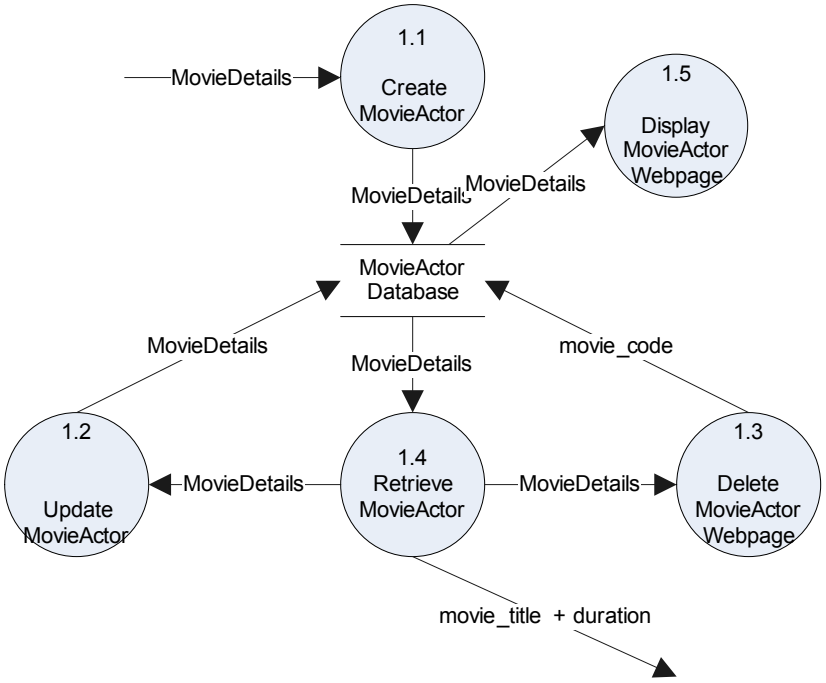

Figure 6.4 Level 1 Diagram – Movie Actor Example – Process 1 (CRUD Movie Actor)

Each subsequent level can be further decomposed with a new level diagram. For example, a level 2 diagram of Process 1.4 could have two processes: 1.4.1 ReadMovieDetails and 1.4.2 FormatMovieDetails.

The last or leave node processes are furthered defined with a module specification, written in pseudo English called Structured English. Typically this specification will use typical programming language concepts, like assignments, loops and conditional statements. For example, the following table shows the module specification for the UpdateActor module.

Table 6.3 Module Description – Movie Actor Example

Name	UpdateActor
Inputs	Movie ID
Description	Adds or deletes actor association with a movie Updates the MovieActor database

On the negative side, DFDs are criticized as confusing to the user, too much detail for requirements, too much detail too soon in the development cycle. It is argued that DFDs can be replace with use cases and class, sequence, state chart, and activity diagram in UML It is the opinion of this author that these criticism is valid if the customer is not technically oriented. In situations where the customer is an internal senior technical staff, this approach can be very efficient and result in well organized project.

6.3.2 Entity-Relationship Modeling (ER)

This data modeling technique is used to provide a conceptual level of abstraction for modeling data structures. ER models are represented by ER Diagrams (ERDs) and have three elements: entities, relationships and attributes. There are many variations of ERD notation but a popular variant is the crow's foot notation used in the example below in Figure 6.5. Entities are represented by a box with a top and bottom part. The top part is the entity name and the bottom part is the list of attributes for that entity. Each entity can have multiple relationships with other entities. These relationships are depicted with a line having specific end symbols. A relationship

denotes how two entities are related by having line drawn between the entities and specifying the cardinality, which is how many of each entity can exists on either end of the relationship. If any entity can have one and only one instance in the relationship, then the symbol is a straight end. If the entity can have more than one, then the symbol is a "crow's foot". A straight bar on the end indicates that the entity is required whereas a small circle indicates that the entity is optional. The example below is from the Movie Actor case study. One movie will have many positions and one actor may be in many positions. Actors and movies can exists without existing in "listed as" (but will not be too successful if that condition lasts for long!)

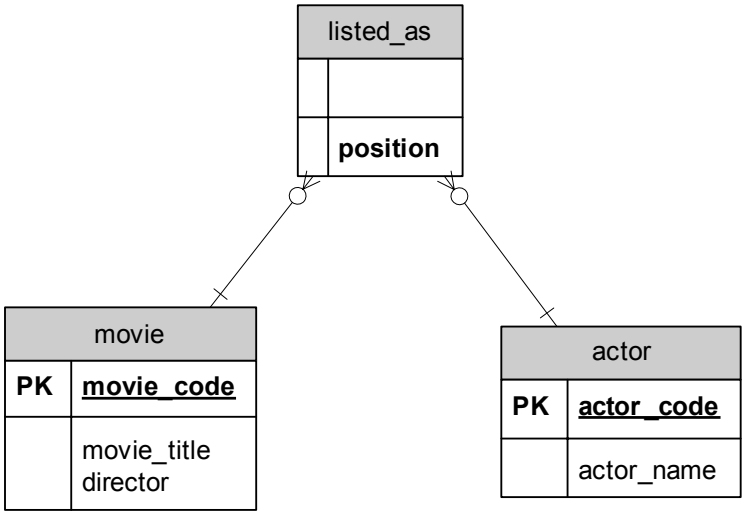

Figure 6.5 Entity Relationship Diagram – Movie Actor Example

ER models lead to creating database objects (tables, indexes, constraints). This author used MicroSoft Visio 2003's Database Modeling Diagram to create the diagram above. However, ERDs do not illustrate any dynamic interaction and must be used with DFDs. The ERD can be confusing to the user, thus being of limited use during the requirements phase.

6.3.3 Structure Charts

In the functional decomposition methodology, structure charts are used to illustrate the high level design, or architecture, of a system. The developer uses this technique to "divide and conquer" a problem by recursively breaking down the problem into parts small enough to be understood. Structure charts are similar to the master blueprint plan that an architect would use to build a house.

To build a structure charts, the main class would be placed in the root of the upside down tree and then place the sub-tasks called by main in the nodes of the root. These new leaves would then have its child nodes representing sub-tasks called by the parent node. This pattern is continued until the "bottom" is hit, ie. no more sub-task. This approach is called "top-down" design and is often followed with "bottom-up" implementation and testing which tests from the lowest level and works its way up building and testing all functionality. [WOLBER]

Figures 6-6 and 6-7 below show the application of this technique to the Movie Actor example.

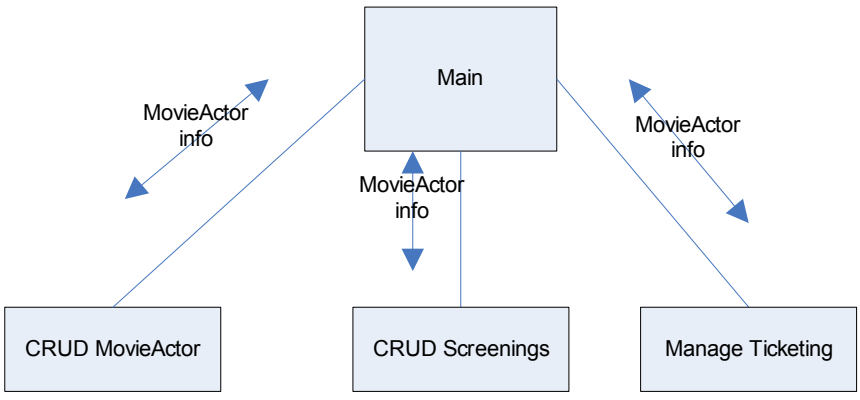

Figure 6.6 Structure Chart – Movie Actor Example

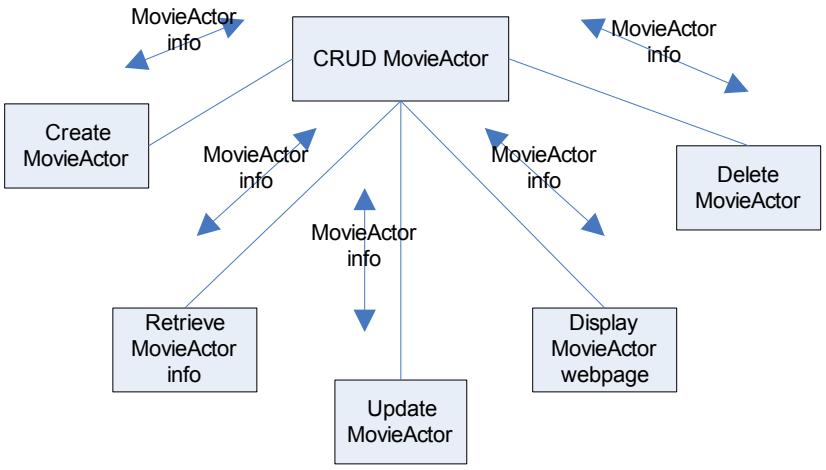

Figure 6.7 Detailed Structure Chart - Movie Actor Example

6.4 Object-Oriented Modeling Language (UML)

"The Unified Modeling Language (UML) is a modeling language for specifying, visualizing, constructing, and documenting the artifacts of a system-intensive process." [ALHIR pg. ix] It provides a suite of diagrams to aid the software engineer to present a system from various perspectives. The goals of the UML are to 1) be already-to-use expressive, simple and extensible visual modeling language, 2) have extensibility and specialization mechanisms for extending, rather than modifying core concepts 3) be independent of programming languages, 4) be process independent, 5) encourage the growth of the object oriented market, 6) support higher-level concepts, 7) address recurring architectural complexities, 8) be scalable, 9) be widely applicable and usable, 10) integrate the best engineering practices. [ALHIR pg. 5-6] The UML is not a programming language, tool or repository specification or a process. Therefore it is wise for the software engineer to be aware of this notation in order to be aware of its potential usefulness when needing to communicate a concept to the customer.

Diagrams are always accompanied with a narrative that is descriptive text that explains the elements of the diagram. This narrative could be notes directly on the diagram or more often a separate page following a standard format. This narrative is written in understandable verbiage and should provide sufficient detail about the general functionality. Pseudo-code snippets can be included in the narrative but it must be able to be understood by any level of expertise. Anyone should be able to read the narrative and flow the diagrams and understand what the system does.

The UML has nine different diagrams, each with a particular purpose: use case, class, sequence, collaboration, state chart, activity, object, component, and deployment. [Kulak pg. 27-33] These diagrams can be can organized into five architectural views : User, Implementation, Structural, Environment, Behavioral. [ALHIR pg. 116-118]

6.4.1 Use Case

The use case diagram is a tool that illustrates the user view of a system and is accompanied with a use case description. Together the description and the diagram create the use case mode. Use case diagrams are the driving diagrams of UML. It is usually implemented first in the development of a suite of UML diagrams. The use case forms a foundation on how the system interacts with external entities: people, other machines, even factors like time, temperature, etc. Use cases have textual description of interaction of external actors and the computer system called the use case document. This tool is used to gather requirements and guide the entire software development cycle. A use case is initiated by an actor which can be a person, computer system, or condition like time or temperature and is represented by a stick figure but is not an

internal entity like an internal database. Actors will interact with a system but not automate it. The role of an actor in respect to the system is denoted by the name given to the actor figure. Role names are useful during requirements and during design to some extent. In the diagram, a use case is symbolized as an oval and had a basic name that describes its function. Actors and use cases are connected with associations (directed arrows) which can be a generalization, extend or include "adornments".

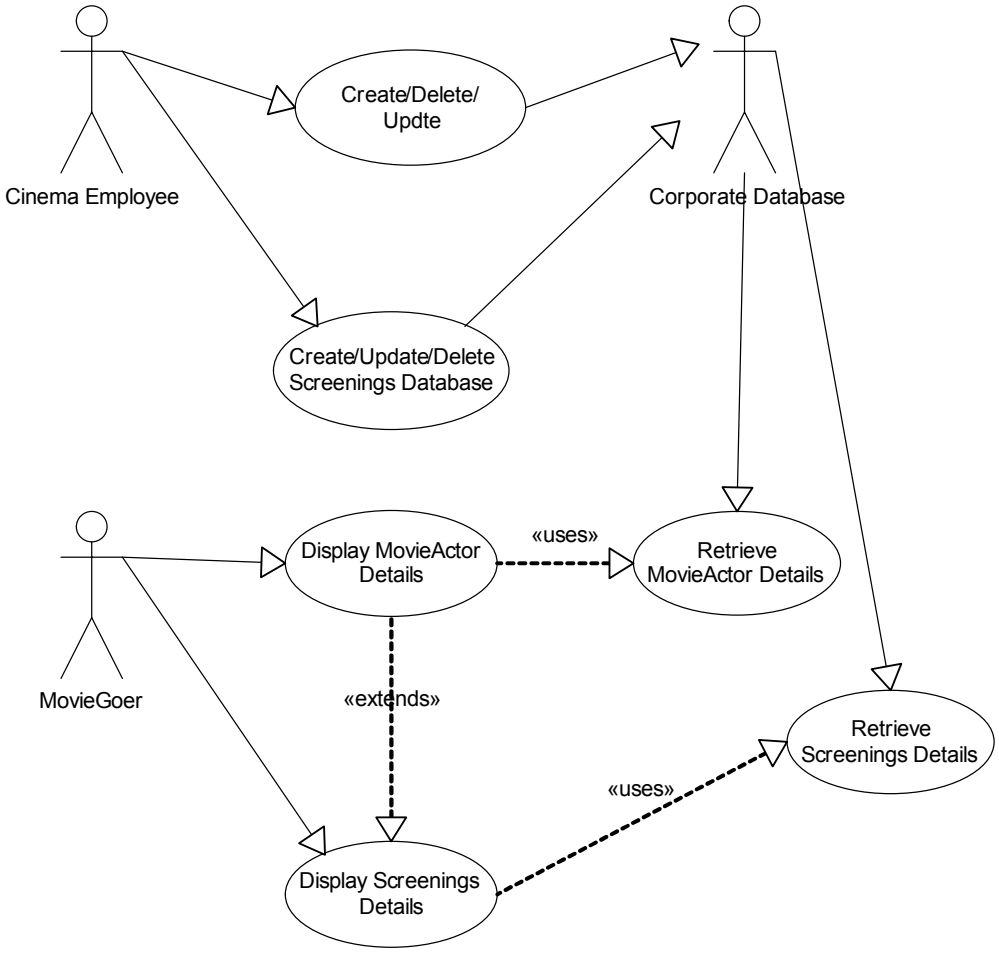

Figure 6.8 Use Case Diagram – Movie Actor Example [MASIACZEK pg. 49]

The accompanying use case document could look like the following.

Table 6.4 Use Case Listing – Movie Actor Example

Use Case List Move Actor Project			
ID	System	Name	Description
U1	Admin	Create/Delete/ Update Movie Database	Allows the user to create, deleted and update the Movie database
U2	Admin	Create/Update /Delete Screenings	Allows the user to create, update, delete the screening database
U3	User	Display Move Actor Details	Displays the reports with the various movie and actor information
U4	User	Retrieve Move Actor Details	Uses the specific input to get the Movie and Actor information from the database
U5	User	Retrieve Screenings Details	Uses the specific input to retrieve the screening info about a movie
U6	User	Display Screenings Details	Displays the various reports with the selected screening details

Table 6.5 Use Case Actor Listing – Movie Actor Example

Project Actors Move Actor Project		
ID	Actor	Description / Use cases
A1	Cinema Employee	Maintains the corporate database with the movie and screening information
		Use Cases: U1 – Create/Delete/Update Movie Database U2 – Create/Update/Delete Screenings
A2	Corporate Database	Stores the actor and movie information
		Use Cases: U1 – Create/Delete/Update Movie Database U2 – Create/Update/Delete Screenings U4 – Retrieves Move Actor Details U5 - Retrieves Screenings Details
A3	MovieGoer	Queries for movie, actor and screening information
		Use Cases: U3 – Display Move Actor Details U6 – Display Screenings Details

Scenarios are used to pick up where the use case leaves off. They focus on detailed interaction. The definition of scenario in requirements gathering is used in at least distinct ways. For the purpose of this paper, a scenario will refer to an example. The iteration value in the scenario is used to specify the current iteration. Daryl Kulak suggests these four iterative values to be used when gathering requirements [KULAK pg. 55]

- façade – outline and high-level descriptions

- filled – broadening and deepening
- focused – narrowing and pruning
- finished – touching up and fine-tuning

These iterations are not to be used as a "lifecycle for requirements" but rather as a way of categorizing activities needed to develop use cases.

The following figure illustrates a scenario for the Move Actor example.

Table 6.6 Use Case Scenario: Movie Actor Example

Use Case Name:	Display Screening Details
Iteration:	Filled
Scenario:	1. MovieGoer Mary pulls up the Screening webpage to see what is playing at the local theaters tomorrow night 2. She enters her zipcode and tomorrow's date and presses Search. 3. The system returns the following information • AMC 2, Pink Panther 2:30 5:00 7:30 10:00 • AMC 2, Curious George 2:00 4:00 6:00 8:00 • Brazos Drive Inn, Curious George 8:00 • Brazos Drive Inn, Nanny McPhee 10:00
Author	Miles Phillips
Date	March 1, 2006

6.4.2 The UML Structural Model View

The structural model view shows how a system is structured rather than how it behaves. These static structure models are also referred to as state models and are expressed in class and object diagrams. A class diagram graphically depicts the classes

and interfaces along with their internal structure and their relationships to other classes. Class diagrams show how classes are constructed and list the names, attributes and operations of a class as well as any associations to other classes statically (generalization and aggregation). Object diagrams are similar but the focus is on runtime instantiations of the classes. They show relationships between objects that are the instances created at runtime from the class templates.

UML defines a class as a description for "a set of objects that share the same specifications of features, constraints, and semantics." [UML2002 pg. 45]. The features refer to the attributes (structural) and operations (behavior). In programming languages, an attribute is called a class member, a member variable, instance variable or field whereas an operation is called a member function or method. When a service provided by a method, a message is sent to an object, as called message passing between objects. A relationship is a meaningful connection between classifiers and are one of several types: association, aggregation, generalization.

An association is specified on classes. However the meaning of a relationship is depicted by the multiplicity notation on each end of the relationship, or the number of instances of one class may relate to one instance of another class. Multiplicity notations include 0, 0..n, 0..1, or 1. If a multiplicity contains a zero, then the participation of that class instance is optional. Therefore, a multiplicity of 0 seems illogical but indicates a unidirectional association.

Generational is a relationship on classifiers, which are classes in the domain class model). It is a special kind of association where an instance of a more specific

class (subclass) is also an instance of a more generic class (superclass). Therefore, multiplicity does not apply to generalization. In contrast, an aggregation is the special association in which an instance of superclass contains instances of other subset class.

The class diagram represents the static structure of a class as well as the dynamic behavior of the system. It is similar to the data flow diagram function used in structured modeling in that both define processes and data structures. Class diagrams focus more on the data structure. If the class diagram does not visualize operations, then it is a pure static structure diagram. The following class diagram is from the Move Actor example. The listed_as class is an associational class used to show the relationship between the actor and movie class. This associational class is necessary because the relationship has its own attribute called position. This example includes both behavior and state features. The first behavior feature of each class is the constructor, a special method to instantiate the objects of the class. The classes also include instance methods with their signatures, ie. the list of arguments and types.

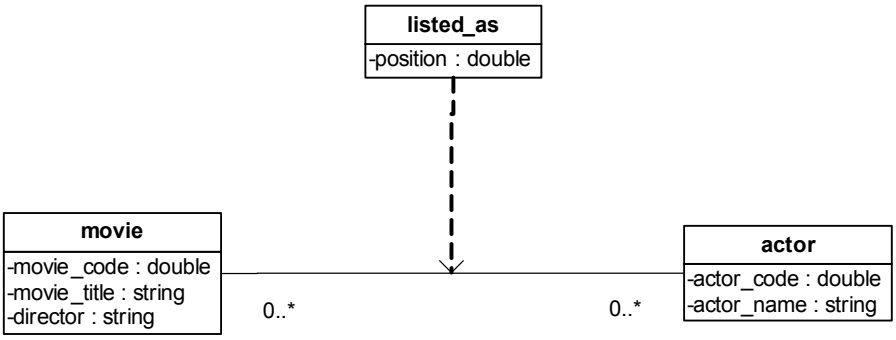

Figure 6.9 Analysis level class diagram – Movie Actor Example

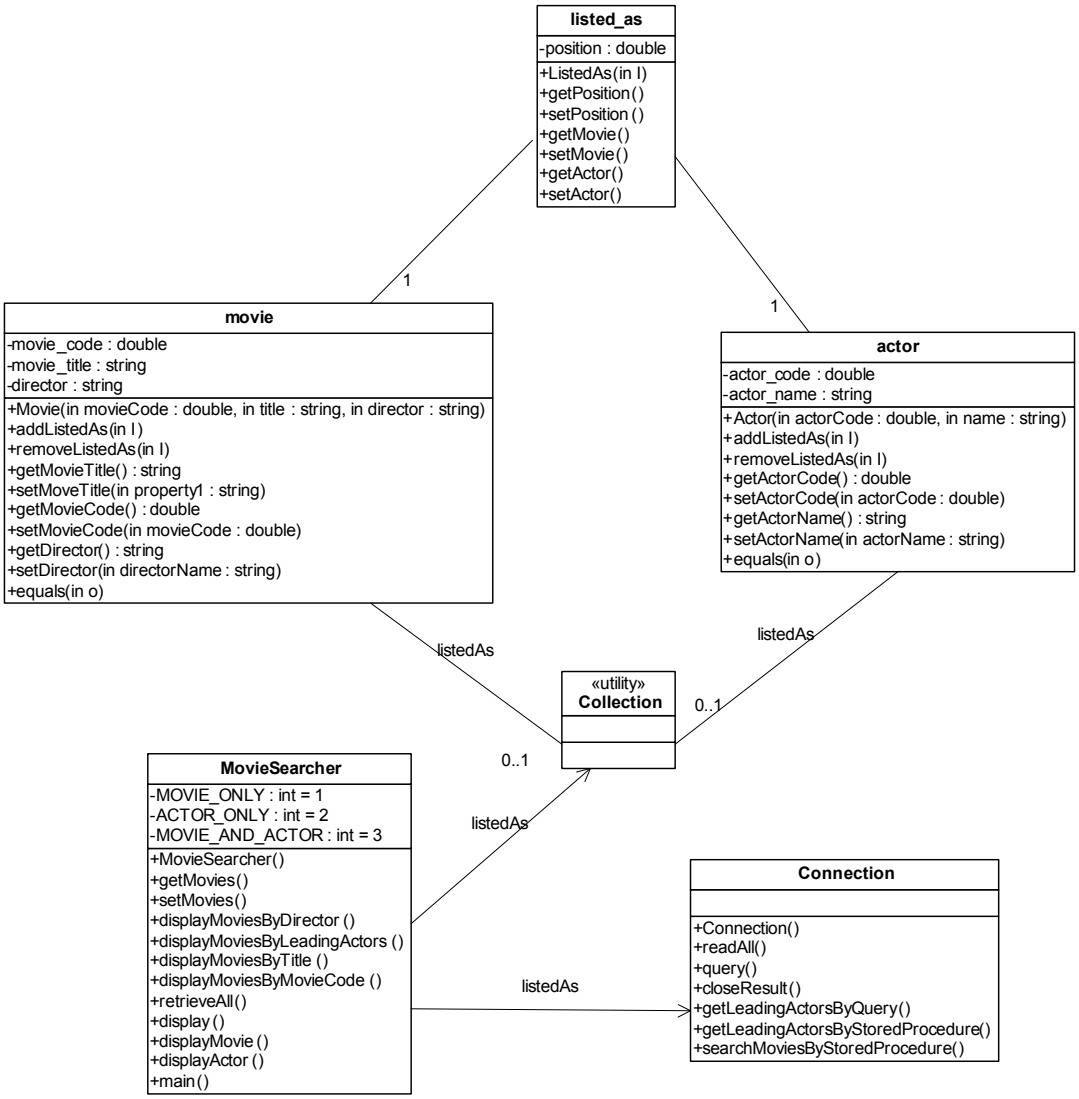

Figure 6.10 Design level class model – Movie Actor Example
[MACIASZEK pg 45-47]

The detail class diagram shows the method parameters and the return code

6.4.3 The UML Interactive Diagrams

The main design level behavior modeling techniques in UML are the interactive diagrams: the sequence diagram and the collaboration diagram. These diagrams are also known as communication diagrams in some literature.

The sequence diagram visually communicates the internal workings of a use case, presenting how messages are passed between objects for a simple, linear relationship. Its name indicates the emphasis of the diagram is to record the sequence of messages. Sequence diagrams have the advantage of presenting more complex models, explicitly displaying message sequences, though they can take up a lot of space.

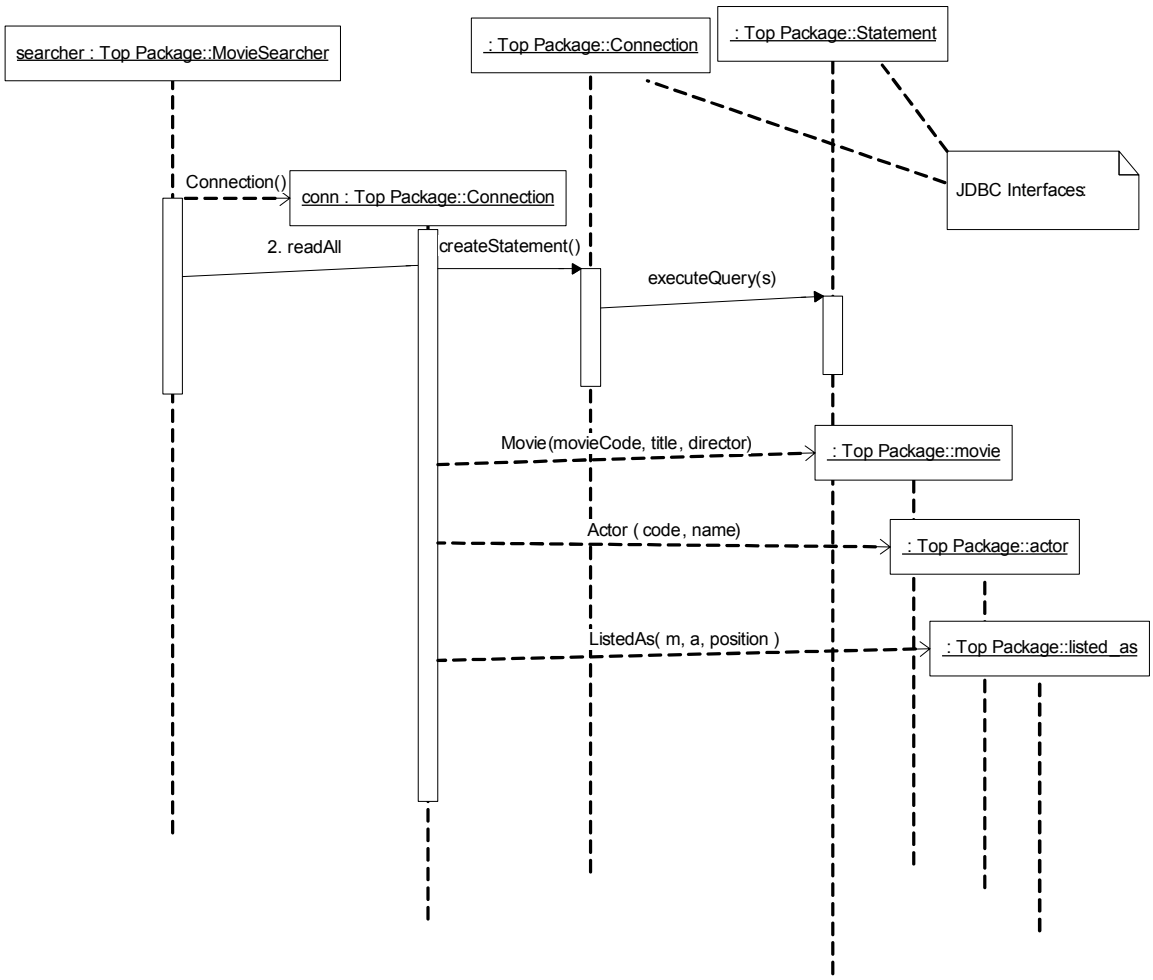

Figure 6.11 Sequence Diagram – Movie Actor Example [MACIASZEK pg 48]

Collaboration diagrams is a lesser used variation of the sequence diagram. These diagrams do not illustrate lifelines like sequences diagrams. Activations and

object creation are implied by the message numbering. These diagrams are useful when analyzing the messages going to or coming from an object. They are beneficial for brainstorming because the ease in which initial draft interactions can be drawn. They can be useful for complex interactions like multithreaded or conditional messaging.

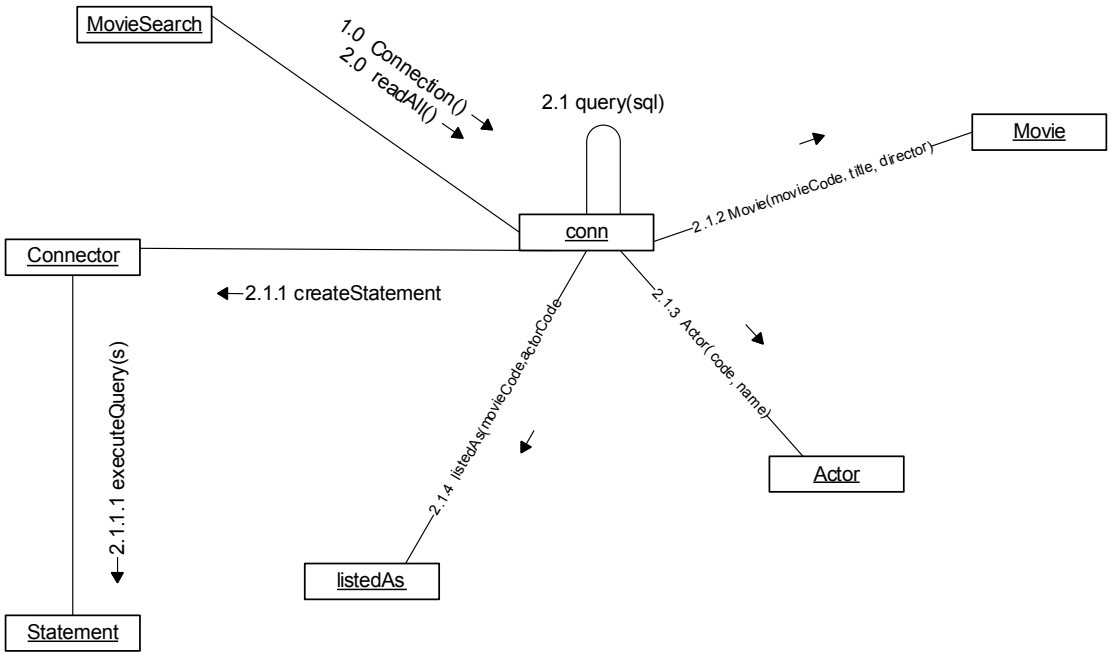

Figure 6.12 Collaboration Diagram – Movie Actor Example
[MACIASZEK pg. 45-47]

These interactive diagrams (behavior model views) show the dynamic aspects of a problem and solution. This view can be referred to as the dynamic, process, concurrent or collaborative view. In contrast, the state charts and activity diagrams focus on how one state transitions to another. State charts are used for simpler views while activity diagrams are used for more complicated views.

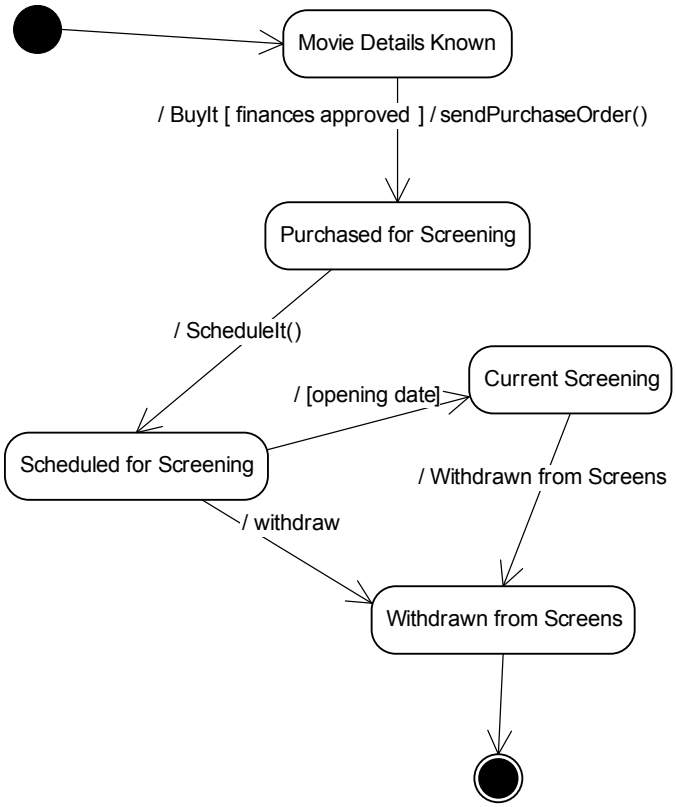

Figure 6.13 State Chart – Movie Actor Example

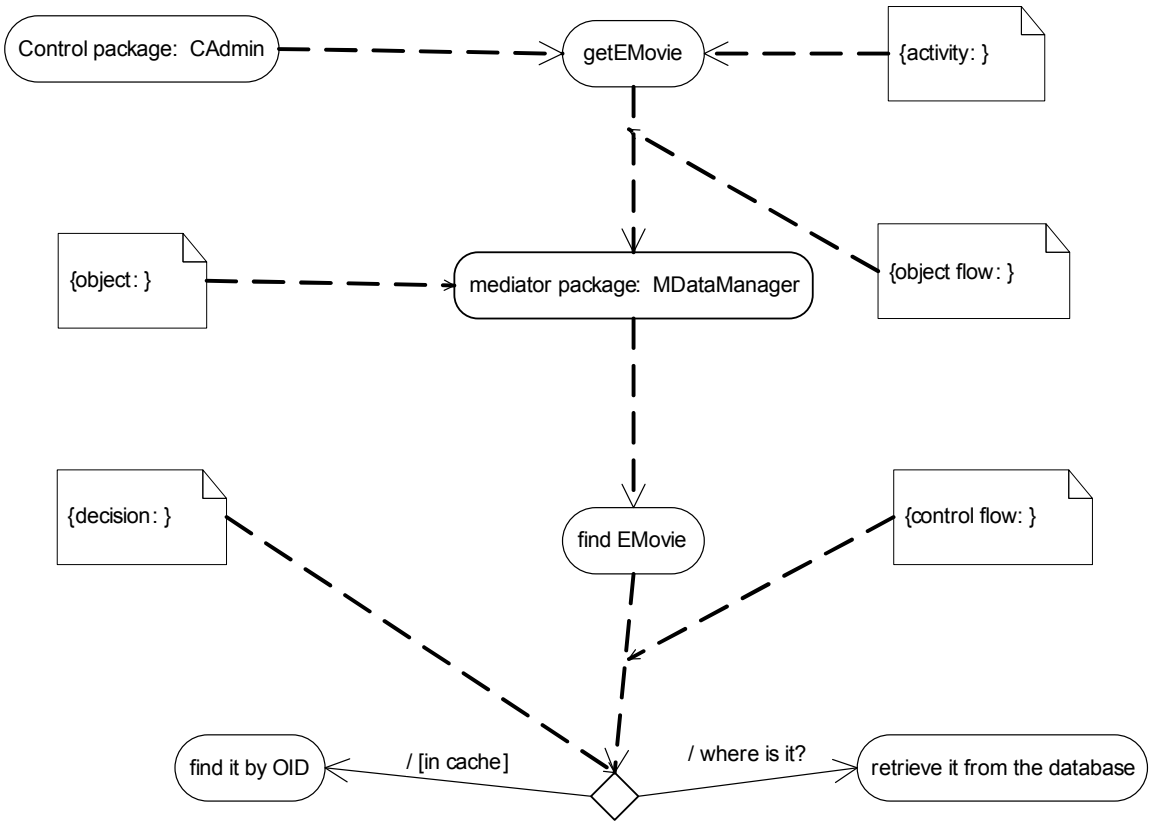

Figure 6.14 Activity Chart - Movie Actor Example

The implementation model view includes both the behavioral and structural aspects of the realization of the solution and is helpful in planning the implementation phase. This includes component diagrams which describe the organization of and dependencies among software implementations. Component diagrams demonstrate how the components work together and help transition from a fine-grained object to a courser-grained object. This diagram helps communicate to the staff responsible for integration and deployment what components will be involved in the installation and now the components relate to each other.

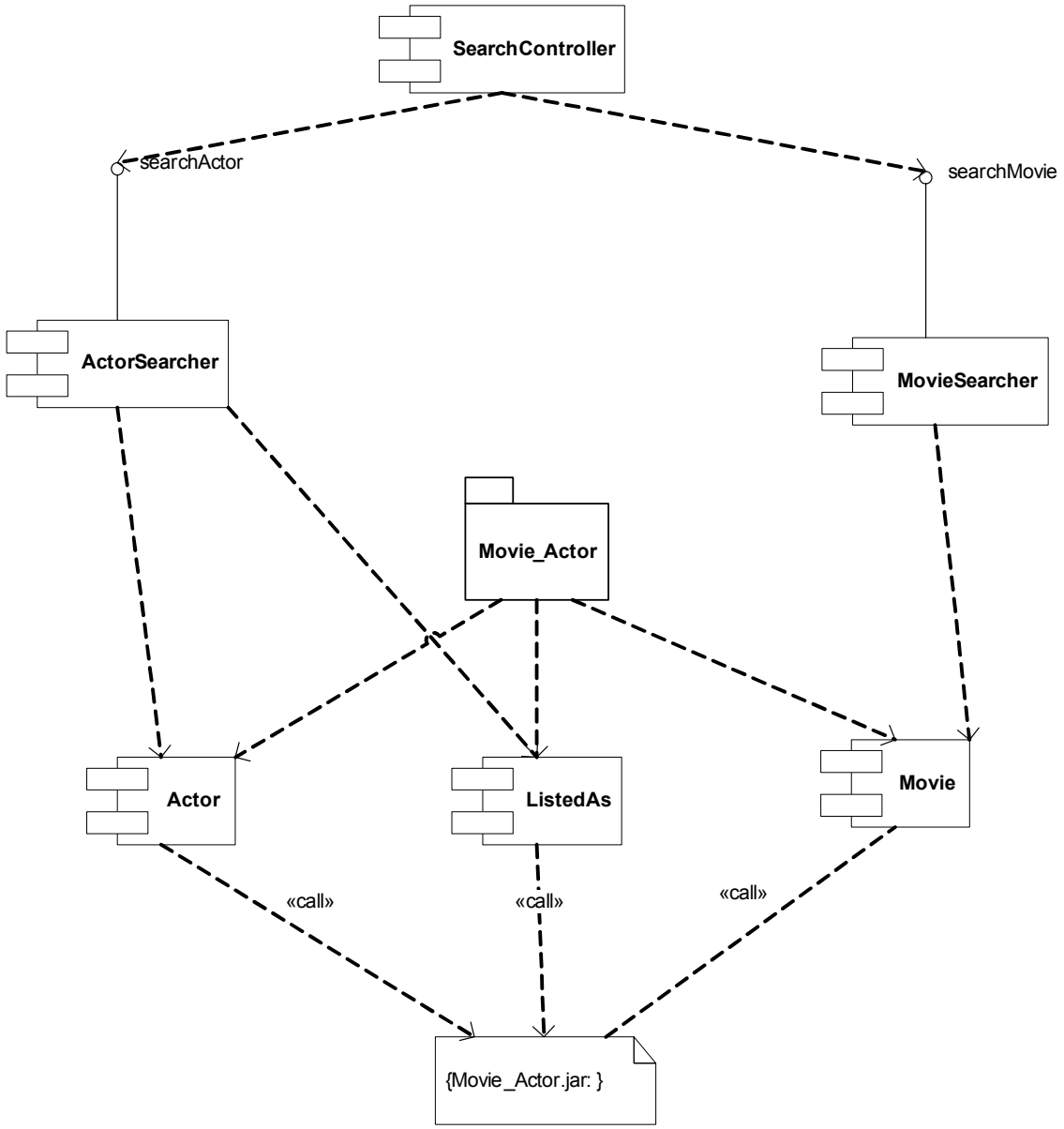

Figure 6.15 Component Diagram – Movie Actor Example

6.4.4 The UML Environmental Model View

The environment model view, also known as the deployment or physical view, has the behavioral and structural aspects of the domain in which a solution must be realized. These contain nodes, components, and relationships. Deployment diagrams show how components will be deployed into the production environment. This

diagram provides a standardize view of the configuration of each component and map how the components interact with each other.

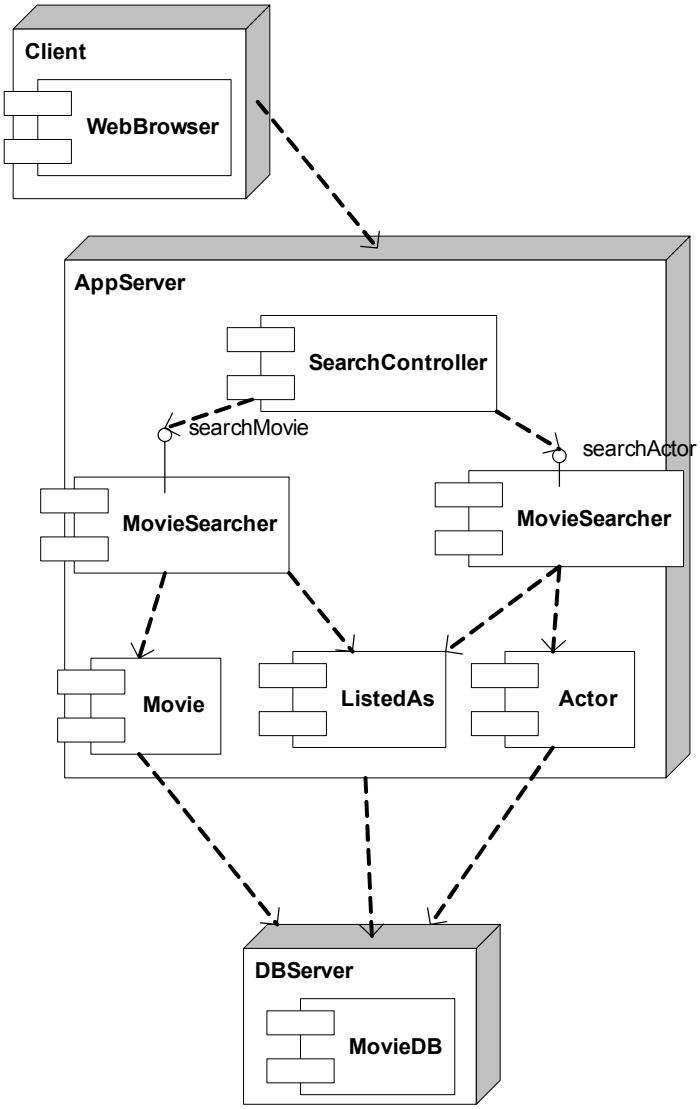

Figure 6.15 Deployment Diagram – Movie Actor Example

Another mechanism in the UML is the package metaphor, which is a way to hide complexity. Packages can be used within use cases, components, or deployment nodes. They provide a graphical method to group components together into one unit. [KULAK pp. 28-33, 116-118]

The advantages to using the UML notation is that it is a well thought out set of tools that can be used throughout the various phases of development to communicate various aspects of the system's requirements and design to the customer, development staff and management. The disadvantages include the learning curve, the lack of directly producing an executable product and it must be implemented and altered to work well within a methodology. Since the UML does not advocate requirements list, scenarios and use cases are used to define requirements and therefore testing and design elements related back to the use case or scenario name. For system with a good number of requirements means that the development team must be willing to generate a large amount of scenarios to capture all the requirements.

For the small software development team the UML has the advantage of providing a cohesive set of documents and diagrams that can be reused for related projects that is readily acceptable by the industry, thus saving time. The smaller the team, the greater the chance that the UML documents will be remembered and reused. The UML also provides a good basis for thought retention (ie. the ability to recall ideas that were presented a year ago.). Therefore if the small development team is properly trained in the implementation of the UML, then this notation provides a consistent

notation through out the life of the project. This author would suggest that DFDs are used to complement the UML in order to show the data flow.

CHAPTER 7

REQUIREMENTS PHASE

Requirements engineering involves learning about the problem and specifying the external or black box behavior of a system that can solve that problem. Traditionally, the requirements phase results in a requirements specification. [DAVIS95 pg. 47] This phase is often the most neglected phase in software engineering, but can also be the most beneficial for reducing the overall lifecycle. Robert Glass states that unstable requirements are one of the top two causes for runaway projects. [GLASS pg. 67] Daryl Kulak and Eamonn Guiney report that they have".. seen more projects stumble of fail as a result of poor requirements than for any other reason." [KULAK pg.2]

Therefore the prudent development team should be committed to properly conducting the activities required to adequately define requirements for each project. Explicitly stating requirements ensures that the user is specifying the functionality rather than the developer. These chapter reviews current software engineering principles that apply to requirements and describes several requirements gathering and definition techniques.

7.1 Requirements Engineering Principles

Certain principles should be followed regardless of the techniques or processes used during the requirements gathering and definition phase. Poor initial requirements

results in invalid cost estimates due to frequent changes, missing requirements, miscommunication, inadequate specification, insufficient analysis.

Understand the problem before writing the requirements. It is important to understand what the problem is before writing the requirements. A small development team should commit to understanding the problem and defining the requirements adequately before diving into design or coding. This can be accomplished by prototyping, user interviews, or collecting data. If all of the requirements cannot be ascertained, then the requirements that are understood should be documented and built, with clear notation that additional requirements will be added in a future iteration signally the customer and management that another round of development will be required.

Fix requirements as early in the development cycle as possible. A team should be committed to fixing any invalid requirements specifications as soon as possible. Studies show that that if a requirements specifications has an error, it will cost five times more to find and fix in design phase, ten times more in the construction phase, twenty times more in the testing phase, two hundred times more if not fixed until after the product is delivered. [DAVIS95 pg. 47] Requirement changes are the most expensive to fix when found during production and the cheapest to fix early in development. [GLASS pg. 71]

Group requirements by functionality and priority. The resulting document of this phase is the requirement specification. This specification can include a prototype of a user interface if applicable. Each requirement should be documented as to why it

was included. Requirements should be grouped by functionality and priority. The priority values can provide information as to what requirements are to be present in each release. This is a particularly important practice for teams committed following the incremental model.

Avoid design in requirements. One principle repeated by various authors when defining a good requirements specification is to avoid design in requirements. Since requirements describe the external behavior of a system, it should not specify the software architecture or algorithm. If necessary, then note design constraints in the requirements so that the designer can generate a design that is within the specified constraints. Requirements should only be concerned with defining the functionalities of the product, not stating how the product will be implemented. This principle should be followed because skill sets of those gathering requirements may not be suited for design or development and solutions must come after the problem has been identified, documented and understood. Also when design precedes requirements, the system tends to take on requirements of its own.

No one technique can be universally applied for all systems. However the requirements are determined and documented, they should be organized sensibly, which usually means hierarchically. Each requirement should have a unique identifier so that it can be referenced easily in design, implementation and test documentation. For complicated systems, it is helpful to use a variety of techniques and viewpoints in order to establish an acceptable level of requirements.

Requirement should be clearly and unambiguously stated. Since requirements are written in natural language (ie. English), they are prone to be ambiguous. A requirements analysis should make every effort to word requirements in an unambiguous manner.

Requirements should be traceable. Design and testing artifacts should be based on requirements, verifying that each design and test element is necessary to fulfill the requirement. Therefore, in order to identify these elements with at least one requirement, the requirements need to be organized in a way that uniquely identifies each one. The design and testing documents can then reference this identifier, clarifying which requirement it is addressing. [IEEE830 pg 4]

Requirements should be verifiable, meaning that within its description some cost-effective process exists that can be used to check if a product meets the requirements. For example, statements like "works well" or "will usually happen" are not verifiable because "good", "well" and "usually" cannot be measured. [IEEE830 pg 4]

Reliability should be specified specifically. Stating a system should be 90% reliable is not any help. Reliability should be specified in measurable terms stating what the percentage of requests that can be incorrect is and what the acceptable down time would be, such as a system should be available 23 hours and 45 minute of every day. [DAVIS95 pp. 47-70]

Missing requirements are the hardest to correct. They lead to omitted logic which leads to persistent software errors. [GLASS pg. 73]

It is important to document what requirements are known early in the project with the understanding that some requirements will be added later. McConnell suggests two approaches to plan for added requirements. 1) expect 80 percent of the requirements to be known before design is started and allocate time for the remaining 20 percent to be introduced later or 2) specify only the top 20 percent of requirements at the beginning and plan for the rest to be developed in small increments. [MCCONNELL2004 pg. 34]

"Requirement analysis is the activities of determining and specifying requirements." [MACIASZEK pg. 16] The determination of requirements is one of the most difficult challenges in software engineering. There is the potential risk that developers misunderstand what the customer requires from the product. "Developers are faced with the following anonymous observation: 'I know you believe you understood what you think I said, but I'm not sure you realize that what you heard is not what I meant.'" [MACIASZEK pg. 16] Therefore the development team should be aware of various techniques used to elicit requirements

- interviewing user and domain experts
- questionnaires to users
- study of existing system documentation
- study of similar software systems
- prototyping of the working model to confirm requirements
- joint application development sessions with the customer

- This leaves the question as to what level of detail is acceptable for the SRS.

Many of the principles listed above can only be quantified by human review. Therefore, testing of these abstract models is difficult and cannot be automated. Walk-through and inspection meetings are popular and effective techniques that are helpful in "testing" the SRS with the purpose of uncovering problem. The emphasis of these meetings is to identify problems, not to find a solution or find fault with individuals. Solutions are left to the requirements engineer to rework the SRS until it satisfactorily defines the product in a way that can be successfully used to execute the design and test activities.

There are a variety of techniques which can aid in communicating requirements between the customer and the development team, depending on the characteristics of the project as specified in the following table. As a project grows in complexity, it may benefit the development team to incorporate several techniques to verify that the requirements are properly understood.

To reduce ambiguity, it is helpful to perform Fagan type inspections on the SRS. This is accomplished by a team of reviewers who are provided with the SRS and a checklist of features that should be included. Each member conducts a review independently and then all meet together to discuss. Fagan's model calls for certain team members to play specific roles in the meeting: a chairperson, recorder, and a reader. All members should be prepared, no management should be present and the meeting should be at least two hours. Any defects found in the SRS should be categorized according to the impact on the resulting project. The result of the Fagan

type inspection is to accept the SRS unconditionally, accept it conditionally or reject it. Inspections are shown to increase the requirements step by 15 to 20 percent but the payoff comes in building the product correctly the first time. Inspections however can find as many as 80 percent of the errors. This coupled with the principle that it is much cheaper to correct a requirement before design cost justifies the effort of an inspection. [MILLERPRESSMAN]

7.2 Requirements Gathering and Definition Methods

Hopefully papers such as this one will serve to educate and motivate developers to commit to good requirements practices. Once a team has committed to requirements, it must decide what how to record the requirements. This section reviews various approaches and the corresponding artifacts that can be used to document.

As stated above, the software requirements specification is traditionally the resulting document for this phase. An SRS should address the following characteristics.

Table 7.1 Software Requirements Specification Characteristics

Characteristics	Question to be answered…
Functionality	What is the software do to?
External Interfaces	How does the software interact with people or other systems or programs?
Performance	What is the acceptable availability or response time, etc.?
Attributes	What are the portability, correctness, maintainability, security, etc considerations?
Design constraints imposed on implementation	What limitations on implementation are required by the customers such as operating systems, database management system, resource limits, etc.?

The IEEEE 830 standard mentioned in the Standards chapter provides a thorough description of the intent, content and suggested format of an acceptable SRS. A good SRS is one that is correct, unambiguous (one interpretation), complete, consistent, ranked for importance and/or stability (essential vs. desirable), verifiable, modifiable, traceable. [IEEE830 pg 4] While some of the characteristics mentioned are obvious, there are a few less obvious characteristics that are worth consideration in the context of this paper. In following the traceable and verifiable principles described above, the method must be stated in concrete and measurably quantities. This IEEEE 830 document further defines the other characteristics in detail. Also since changes to requirements are inevitable, the document must be written and organized in a modifiable manner regardless of the format chosen. This document should include the services that the resulting system will provide and also the constrains on the system such as the "look and feel" of the user interface, acceptable performance criteria, security limitations, or any operational, political or legal considerations. Some of the challenges to requirements gathering are

1. finding out what the users need
2. documenting user's needs
3. avoiding premature design assumptions
4. resolving conflicting requirements
5. eliminating redundant requirements
6. reducing overwhelming volume
7. ensuring requirements traceability. [KULAK p 11]

The industry provides many formats of SRS, each with its own advantage. This paper does not attempt to cover all the available formats. The following methods are presented to expose the reader to some possible approaches for gathering and defining requirements.

7.2.1 IEEEE Software Requirements Specification (SRS) 830-1998

For years, software development teams have used templates such as the IEEE 830 Software Requirements Specification document to detail all requirements of a project. Requirements are gathered by interviewing the customer, examining the current work flow, wording the requirements in an unambiguous manner and then attempting to lock the customer into agreeing that this SRS completely defines the scope and goal of this product. In this ideal scenario, the development team can then develop the perfect product after months of isolation. Even though this scenario is never realistic, the traditional requirements list operations under that assumption. The following is the description of the IEEE SRS.

"The SRS is based on a model in which the result of the software requirements specification process is an unambiguous and complete specification document. It should help

1. Software customers to accurately describe what they wish to obtain;
2. Software suppliers to understand exactly what the customer wants;
3. Individuals to accomplish the following goals:
    a. Develop a standard SRS outline for their own organizations
    b. Defined the format and content of their specific SRS

c. Develop additional local supporting items such as SRS quality checklist, or an SRS writer's handbook." [IEEE830 pg. iii]

The IEEE document goes on to state that an SRS should provide the following benefits: agreement on what the product is to do, reduce the development effort, provide a basis for cost estimation, make it easier to transfer the product to new users, provide a base for enhancements. It would be helpful to state how IEEE defines the following terminology as related to an SRS: contract (legally binding document agreed upon by the customer and supplier), customer (the person, or persons, who pay for the product and usually decide the requirements), supplier (the producer of the product for the customer), user (the person, or persons, who operate or interact directly with the product. [IEEE830 pg. 3]

The IEEEE standard encourages the use of prototyping to demonstrate the characteristic of a system. This practice is helpful because the customer will usually react to a prototype than actually read an SRS, facilitating quick feedback. A prototype will also expose unexpected aspects of the systems, raising new questions about the requirements of the system. Using a prototype also tends to shorten the change during development and is used to elicit requirements. [IEEE830 pg. 9]

The SRS should not include partitioning the software into modules, allocating functions, describing the follow of information or control, or choosing data structures. However, the SRS does provide for the documentation of design constraints like physical and performance requirements, software development standards and software quality assurance standards.

IEEEE Standard 830-1998 suggests the following format for a standard SRS

1. Introduction
    1.1. Purpose
    1.2. Scope
    1.3. Definitions, acronyms, and abbreviations
    1.4. References
    1.5. Overview
2. Overall description
    2.1. Product perspective
    2.2. Product functions
    2.3. User characteristics
    2.4. Constraints
    2.5. Assumptions and dependencies
3. Specific requirements
    3.1 External interface requirements
        3.1.1 User interfaces
        3.1.2 Hardware interfaces
        3.1.3 Software interfaces
        3.1.4 Communications interfaces

3.2 (*see below)*

3.3 Performance requirements

3.4 Design constraints

3.5 Software system attributes

3.6 Other requirements

Appendixes

Index

[IEEEE830 pg 11]

Each component is described in the specification standards document. Section 3 should contain a description of all specific requirements in sufficient detail as to allow the developers to design a system to satisfy the requirements and to allow testers to test that the system satisfies the requirements. Each requirement mentioned should be observed externally. At minimum, the requirements should include every input and output of a system. Requirements are composed of the following items: External interfaces, functions, performance requirements, logical database requirements, design constraints. Functionality are fundamental actions (usually starting out with "The system shall …") and include input validation, sequence of operations, responses to abnormal situations, relationship of output to inputs (formulas). Also included in the SRS are software system attributes such as reliability, availability, security, maintainability, and portability.

The IEEE document [IEEE830 pp. 18-26] describes various organizational methods of the system requirements:

1. By system mode – such as training, normal, emergency and is either organized by globally identifying the external interface, functional, performance and requirements and listing the modes by functional requirements or by specifying the

external interfaces, functional and performance requirements within each mode. Reference IEEE 830 Annex A.1 for more details.

2. By user class – software provides a different set of functions to different classes of users (administrator, "normal user", super user) . Each functional requirement has a list of classes. Reference IEEE830 Annex A.3 for more details.

3. By objects – group real-world entities together as classes and define each class a set of attributes and functions (services, methods, processes). Reference IEEE830 Annex A.4 for more details.

4. By Feature – usually a sequence of stimulus-response pairs. Reference IEEE830 Annex A.5 for more details.

5. By Stimulus – group the functions by stimulus (activation event). Reference EEE830 Annex A.6 for more details.

6. By Function hierarchy – to be used when none of the methods prove helpful. The functions are organized by common inputs, common outputs, or common internal data access. This format references data flow diagrams and data dictionaries. Reference IEEE830 Annex A.7 for more details.

7. By multiple organizations – useful when combining user class and feature. Reference IEEE830 Annex A.8 for more details.

Some advantages of using the traditional SRS format are

- inclusion of all requirement elements
- standardized so that all projects have similar look and feel
- some flexibility for multiple perspectives of the requirements

Criticisms to using the traditional SRS format are

- usually written in natural language (like English) which is inherently ambiguous. If used, each requirement should have its description reviewed by a third party for clarity.
- hard not to embed design into requirements
- too little or too much effort
- prototypes are not useful or are distracting
- difficult to use
- difficult to alter or rearrange
- no checks and balances with other systems
- easy to write a duplicate or obsolete requirements
- some alternatives like the functional hierarchy could easily result in 100s of pages to list all the data elements in this format.
- not usually references after the requirements definition activity is completed so why bother?

7.2.2 Requirements modeling with UML

Traditional techniques of expressing requirements functionality include the requirements specification, data-flow diagram (DFDs), entity relationship diagrams (ERDs) and prototypes. While these traditional techniques have served the industry well for several decades, some believe that more modern object-oriented approach is superior. Kulak and Guiney feel that these traditional requirement specifications are often not used after they are produced and therefore are declining in their usefulness.

They contend that requirements list are full of duplicate or conflicting requirements and do not provide the user with a cohesive view. ERDs and DFDs help move into programming an database design but are confusing to the user. Prototypes are helpful to the user but take the focus off of the system requirements and on to the user interface. Therefore, Kulak and Guiney feel that these techniques not be used by the requirements analyst. DFDs can be replaced with use cases and class, sequence, state chart, and activity diagrams in the UML. ERDs can still be helpful but not as a user communication tool. Kulak and Guiney encourage the use of use cases to define requirements in their book appropriately named "Use Cases Requirements in Context". [KULAK pp. 20-21]

Requirement should be gathered iteratively and incrementally in order to reduce risk by treating at risk items early in the development cycle. Since requirements specifications change constantly due to the fact that requirements are based on people's fuzzy ideas. Kulak breaks down the iterations into four logical steps (or mindsets) with each step further defining the requirements. He gives these steps the following names and descriptions:

- façade – outline and high-level description
- filled – broadening and deepening
- focused – narrowing and pruning
- finished – touching up and fine-tuning

The steps are not to be viewed as a lifecycle for requirements but rather of a way to categorize activities needed for use cases. Throughout the iterations, several "tools" are developed to define the requirements deliverable set. These "tools" are

- problem statement – description of the business problem to be solved. This statement should be written by the high-level executives who are approving the need to solve the problem.
- statement of work – defines the scope of the work and general view of how the work is to be accomplished. It includes items such as scope, objectives, application overview, user demography, constraints, assumptions, duration, etc.
- risk analysis – ranking the risk associated with each state
- prototype – software mockups of a system's user interface
- use cases and use case diagram – defines and visually describes the system's interaction
- business rule catalog – written and unwritten rules that dictate how a company conducts its business

Using use cases to gather requirements is all about iterations. The user views the system as black boxes so requirements documents should put everything in context of "going in and out" of these boxes. Then the next iteration the box expands and the next step is detailed, calling another set of black boxes.

7.3 Formal Methods

Formal methods provide an approach to removing ambiguity in requirements but are difficult to understand and implement, therefore making it not usually desirable

for the small development company. Since small projects will usually yield relatively small requirements documents, then a reasonable effort to word requirements unambiguously within the natural language is usually appropriate. If it is decided to incorporate formal models of requirements into the SRS, then include the natural language description on the facing page so that the intent of the formal model is easily identified.

Implementing formal methods requires a high learning curve but if a team has members that are trained and are well versed in this approach, then it can be a very efficient way to define requirements as long as there is sufficient attention to provide an accompanying user level explanation.

7.4 FREEDOM Method

Traditionally requirements have defined as "what the system shall do but not show it should do it" " a condition or capability that must be met or possessed by a system to satisfy a contract, standard, or other formally imposed document" [LUTOWSKI pg. 20] The practical use of this definition is limited because it is too ambiguous to offer guidance. The Freedom process (used by the NASA team for developing software for the Space Shuttle Freedom) provides a more useful perspective of requirements, consisting of

1. all external stimuli of system
2. all associated external responses
3. all external communication protocols [LUTOWSKI]

This stricter definition provides a litmus test for the customer and developer to use in determining what is a requirement and what is a design or implementation constraint. Ideally the customer should not specify design or implementation constraints when determining the requirements of a system. The customer should work with the designer to establish first the stimuli and then the associated responses to the stimuli.

In order to understand what is truly a requirement, it is helpful to define precisely what elements are to be composed in the design specification. Lutowski states that a design specification consists of the following

1. identification of each module block box and encapsulated info
2. identification of relationships among the module black boxes
3. specification of module stimuli
4. specification of response behavior of each module stimulus
5. specification of detailed communication protocols that comprise the stable interfaces to the modules

[LUTOWSKI pg. 23]

The Freedom requirements process is illustrated in the following figure.

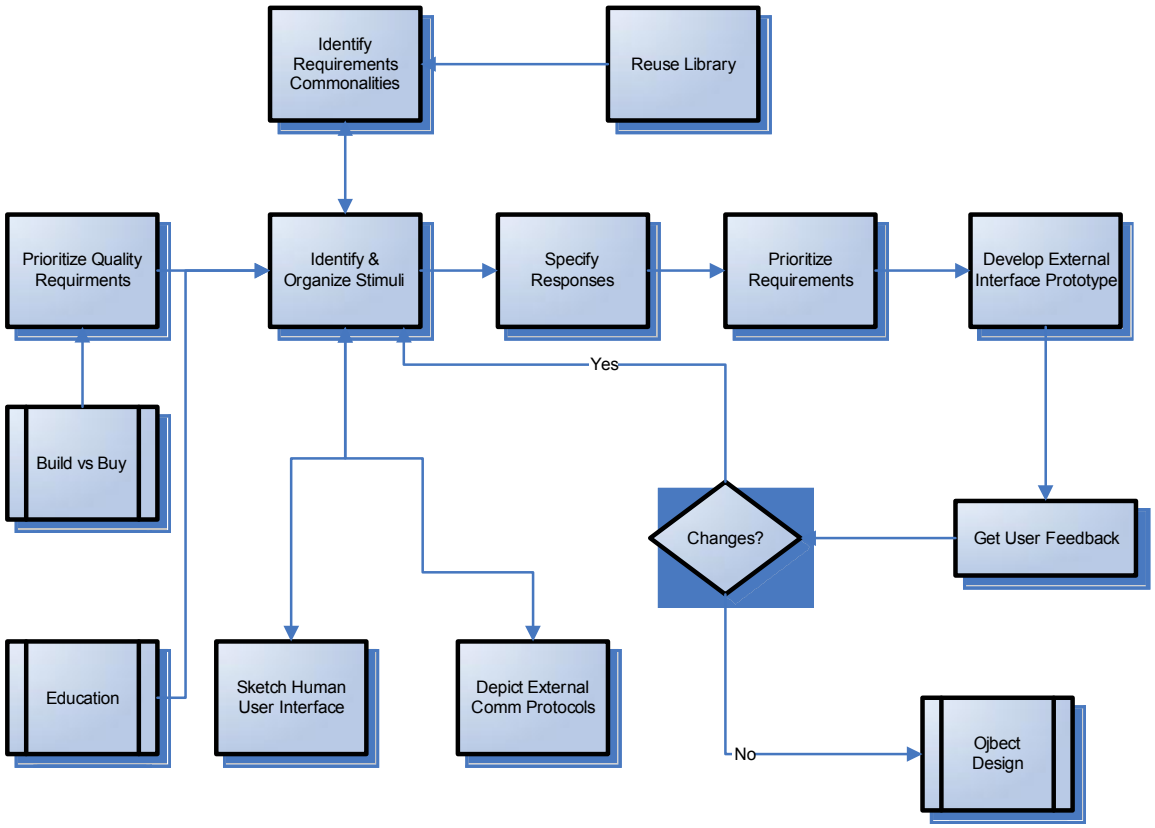

Figure 7.1 Freedom Requirements Process [LOTOWSKI pg. 44]

This process is not a step by step procedure that must be rigorously followed but documents a rational process established by Parnas and Clements. [PARNAS] Lutwoski suggests that steps can be "faked" by producing the required products. The process will work regardless how the products are produced since requirements encapsulation and other aspects of Freedom rely on these notations. [LUTWOSK pg. 31] Each Freedom task is described below

- Build versus buy – requirements process is to be started only after the customer determines that no off the shelf product can be purchased to satisfy the need.

- Prioritized quality requirements – the customer should modify the default quality ranking of each requirement. Once generated the list of customer ranked quality requirements should be publicized and emphasized to all team members.

- Identify and organize stimuli – Stimuli are identified for each input source in the context diagram and put on the functionality tree which organizes stimuli based on their activation. This task is central to the requirements process.

- Sketch Human User Interface The development of functionality screens help define stimulus. On the sketch, stimulus manifest themselves as buttons, data entry field, lists, etc.

- Depict external communication protocols. Specifying the communication protocol for accessing files or external database of external systems is helpful in identifying stimulus.

- Identify requirements commonalities. Repetitions or commonalities in the functionality trees should be identified and given a reference name. This activity will lead to reusable requirements components, simplifying the functionality tree, application code and future applications.

- Reuse Library. The library of reusable components should be inspected once a requirement is known. If the requirement specification exists in the library, then a simple reference name is recorded on the functionality tree.

- Specify response – Behavior tables should be created using the Program Design language to describe the externally detectable responses of each stimulus.

Design and implementation constraints are allowed in the behavior table but should be should be segregated from external responses of true requirements.

- Prioritize requirements – the customer may prioritize the requirements based on the importance or urgency of the need by annotating the functionality tree. These annotations provide the selection criteria for incremental releases. For non-incremental development models, then there is no need for prioritization since it will be all considered in the current release.

- Develop external interface prototype. Developing an internal mockup of the GUI is refined by depicting the interfaces to external systems and environment. This prototype is useful in obtaining customer feedback early on in the process.

- Object Design. The resulting artifacts of the requirements process are object designs which will serve as the input of the object design process. This document specifies the capability requirements from the functionality tree, associated behavior tables, and functionality screens.

Freedom process categories requirements into two types of requirements: capability and quality [LUTOWSKI pg. 43-44]. Quality requirements must be determined first to define the scope and the capability requirements are developed using the quality requirements. Capability requirements also called "functionality requirements" are recorded in the stimulus-response behavior of the external system interface. The quality requirements are those measurable attributes of the system as a whole including a ranking by the customer on the importance to the customer.

Kuluk's book "Use Cases Requirements in Context" provide some excellent examples of how to develop a functionality tree and the related behavior table. Examples of these techniques were not included. This book is recommended for review if this approach sounds applicable to a team's project.

The author of this paper found the Freedom approach attractive for the specification of requirements of an interactive intense application. The simplicity of the artifacts allows for ease on updates while providing a comprehensive format to specify the expected reaction to each stimulus. It is interesting to note that the practice of requirement encapsulation could reduce maintenance costs by 20 percent. [LUTOWSKI pp. 230-232] Statistics like this entice the exploration of this alternative for a team of any size.

7.5 Reasons Why Requirements Are Not Adequately Defined

With all this convincing evident and variety of methods, why would any well trained software developer not generate acceptable requirements documentation? Here are some of the common reasons why development teams neglect writing requirements.

- takes too long
- documents wrong thing
- makes assumption about activities that have not happened
- often completed in time to do it again
- not as fun as programming

The development team can go off track with requirements when the SRS includes embedded design considerations, is vaguely written, or computer science

jargon is used so that the customer does not fully understand the requirement. Common problems with the requirements gathering and definition techniques are

- Developers tend to get ahead and start embedding design into requirements
- Either too little or too much effort is spent on requirements
- Prototypes are useful but are also distracting
- Requirements lists that are difficult to use and do not provide any checks or balances

Regardless of the technique used to gather and document requirements, the small development team should be committed to developing acceptable requirements at the appropriate detail and in the format that is useful for that project.

7.6 Case Studies

[PFLEEGER pp. 69-71]

The following example demonstrates the use of the Lai notation to identify and quantify risk in the project within the boundaries of the spiral process model. In choosing the process model for the Piccadilly television advertising program, several options were considered. The chief constraint was the type of advertisements that can be sold and when it can be displayed, and that regulations can change with Advertising Standards Authority rulings. The waterfall model was too rigid since it provided little flexibility after requirements analyst stage is complete. Some prototyping was included to build the user interface but not the primary process model. Since most of the uncertainty of the regulations and business constraints, a process model was needed that can used and reused as the system evolves. A variation of the spiral model was chosen

for the Piccadilly system because its inherit nature of reviewing assumptions, risks and prototyping various characteristics. The repeated evaluation of this model has the necessary flexibility to change the requirements and design. The original Boehm spiral model does not have the necessary detail to guide analysts, designers, coders and testers. [BOEHM] Therefore, other techniques and tools were used for finer levels of detail. Lai's notation was used to represent risk as an artifact to measure and track risk with each iteration. Risk was divided into probability (likelihood that a particular problem will occur) and severity (the impact it will have on the system). For example, one element of risk is not enough training in the development method. If object-oriented approach is chosen, then the developers must be trained in this practice. This risk the system not properly implementing the object-oriented system is lowered by requiring all developers to attend a four week object-oriented course. However, if this training is not sufficient, then the project would probably not be completed on time. The probability is low but the severity is high. . This risk situation could be evaluated using the Lai artifact table technique.

CHAPTER 8

DESIGN PHASE

Design is the set of activities in which the architect is defined that will satisfy the requirements and also the specification of the algorithm used to by each software component. "The architect includes specification of all building blocks of the software, how they interface with each other, how they are composed of one another, and how copied of components are instantiated and destroyed." [DAVIS95 pg. 73.] While requirements describe the functions and/or attributes that are externally visible, a design describes a subcomponent of a system and/or its interfaces to other subcomponents. [IEEE830 pg . 9]

During the design phase, certain principles should be followed. The small development team should realize that transitioning to the design phase from requirements is a conscious and difficult task. It is the process of converting an external view of the system into an internal view and therefore should be allotted a significant amount of the project's overall time. Some obvious, but not always applied, design principles that need mentioned but not elaborated are use of efficient algorithms; make modules efficient, flexible and general. "Great designs are clean, simple, elegant, fast, maintainable, and easy to implement. They are the result of inspiration and insight, not just hard work or following a step-by-step design method." [DAVIS95 pg .95]

"When moving from requirements to design, there is an explosion of 'derived requirements' (the requirements for a particular design solution) caused by the complexity of the solution process. The list of these design requirements is often 50 times longer than the list of original requirements." [GLASS pg. 76] How well a system is designed determines the quality of the product. "Efficiency stems more from good design than from good coding". [GLASS pg. 139]

One of the long standing debates in software engineering is how much effort should be spent on design. In ten years, the trend had changed from design everything to design nothing. It is this author's opinion that each project would benefit from some level of design, however it is accomplished or recorded. If any activity in this phase needs compromised, then it should be in making the documentation pretty. Time spent hashing through various design alternatives is well worth the effort. While polished documents of bad or mediocre designs might be impressive on the surface, it does not help to efficiently create a product.

8.1 Design Principles

The following principles represent the best of the modern software engineering principles that relate to the design phase. Designs that include these principles should be result in products that are stable, robust and usable.

Conduct design reviews. A good review of a product will require about half of the time that it took to produce it. This same ratio applies to all phases of the software development (design / design review time , requirements / requirements review time, etc.) [HUMPHREY pg. 35] While this ratio initially seems greatly disproportional,

design review is critical for identifying problems in the design. Writing test plans as designing is good way to combine review efforts, review the design for defects while discovering functions to test.

Enforce standardize configuration management. Each team should have a standard naming convention for design components and documentation as well as a standard method of storing these artifacts. This system will not be used effectively if it is difficult to use , is not available when needed or produces unexpected results. It is worth the investment to provide quality tools for this function. This author uses Surround SCM by Seapine Software for this purpose with satisfactory results. Each team member should be aware of the naming conventions for storing source code and related artifacts.

Traceable to requirements - As with requirements, there is no design methodology that is optimal for every project. Whatever methodology is used, the design components must be traced back to requirements. This principle will assure that all requirements are addressed and that each design component has a valid reason for its existence. This identify will also link the design components to the test plans since both are referencing the same requirements identifier.

Create proper design documentation. - Design is manifested in documentation. Without proper documentation, the design does not really exist. Therefore, a development team should be absolutely committed to the appropriate level of design documentation for that project to facilitate good communication within the team. Granted, a small development team should have daily, even hourly contact with

each other but design documentation is still necessary to assure that the design provides solutions for all the requirements in such a manner that is acceptable to the customer. Providing adequate documentation also will assist in preserving ideas in case a project gets put to on hold.

Encapulsate design components. - The principle of encapsulation applies to design as well as requirements and implementation. Information hiding is a proven, simple concept which makes software easier to test and maintain. Encapsulation refers to a uniform set of rules about which types of information can be hidden. [DAVIS95 pg . 70] Following this principle leads to a modular design which is easier to maintain and enhance as new or modified requirements are introduced.

Create reusable components - A small development team should be able to reuse software components and reference these in design documents. This principle allows a team to build on it past success. In order to transition a team from reinventing the wheel for each project, a thoughtful and systematic process should be agreed upon which will allow all members of the team to access and store consistently build and documented software components. Following this principles will allows team members to be consistent in following coding conventions which should make the code more manageable. As a component is designed, the developers should make the interface and name as general as possible so that the component can be reused easily under a variety of conditions without having to modify the component. While design reuse is not a new concept, it gained popularity in the 1990s when it was packaged in “designing pattern”. A design pattern is defined as “a description of a problem that occurs over and over

again, accompanied by a design solution to that problem. A pattern has four essential elements: a name, a description of when the solution should be applied, the solution itself and the consequences of using that solution." [GLASS pg. 56]

Design for error detection. - Since errors are unavoidable, the software engineer should design a product so that errors are easily detectable and that undetected errors are not critical. Some ideas that aid in following this principle are never assume a value for a case statement (ie. always include a default case), develop strategies for as many "impossible" scenarios as can be imagined and do fault tree analysis to predict unsafe conditions. Proper error handling should not allow "garbage in, garbage out". Inputs that would be considered "garbage" should not be processed but raise a meaningful error condition.

Aim for low coupling and high cohesion. – In order to design to easily detect errors, the architecture or algorithm should be written in a simple manner. Two ways to implement this principle are low coupling and high cohesion. Coupling is the measure of interconnection among modules in a software structure. The less complicated the connection, the lower the possibility for a negative ripple effect to related modules. Cohesion is the measure of how clearly defined a particular module is. A module with high cohesion does one or a few things well. Design should have low coupling and high cohesiveness, manifesting in these properties of a system: [WIKIPEDIA COUPLING]

- a change to one module does not usually require changes to other modules in the system
- source of errors is easy find

- new requirements are easy to implement.

Spend adequate time on design. As a rule of thumb, every hour spent in coding should be matched with a least an hour of design with a design time to coding time ratio of 1 to.5 being optimal. If more time is spent coding than in design, then most likely a significant about of design is being conducted during coding. "Because developers typically inject more than twice as many defects per hours in coding, designing while coding is not a sound quality practice." [HUMPHREY pg. 35]

Design for maintenance - Experience shows that the most expense phase of a product is the maintenance phase. The architectural selection directly affects a product's maintainability. This principle is often neglected during design, thus increasing the maintenance cost.

Design multidimensionally. A good design is represented in many ways to fully understand and convey its essence to the customer and the development team. The complete design should include the following representations.

- Packaging – a hierarchy chart representing "what is part of what?"
- Needs hierarchy – a network diagram of components with arrows indicating which components need something. Represents "who needs whom?".
- Invocation – a network diagram of components with arrows indicating which components call, interrupt, or send messages to others. Represents "who invokes whom?".

Process – set of components are packaged together as asynchronous processes and specified as to what conditions would cause the process to be created, executed, stopped and destroyed. [DAVIS95 pg. 94]

Use design patterns. Solutions for many of software's most common problems have been captured in design pattern (design reuse). The book *Design Pattern* [GAMMA] extensively describes several patterns. This paper does not include many details about design patterns due to the level of detail required to provide an adequate review of these pattern. Each developer is encouraged to examine this book to evaluate if design patterns will assist his or her design efforts. Patterns can aid design by

- Reducing complexity by providing ready-made abstractions
- Reducing errors by providing well thought out solutions
- Reduces time by providing a list of alternatives instead of having to create from "scratch".
- Improve communication by allowing the design dialog to be at a higher level.

8.2 Design Practices

Iterate. Design is an iterative process. As the design evolves, it requires revisiting the previous design components. Continuous review of the design from high-level to low-level provides a healthy dynamics to assist in defining a stable design.

Use both top-down and bottom-up approaches. By applying both composition and decomposition approaches to a solution, strengths and weakens of the design are more easily captured.

Experimental prototyping. Some designs cannot be proven as viable without implementing it and therefore requirement some level of prototyping, which presents a paradox of having partially solve a problem before completing the design. Prototyping can provide an expensive way to address this problem by writing the absolute minimum amount of throwaway code that is needed to answer a specific design question. The research question should be narrowly defined to improve the chances of the prototype being helpful. Several authors warn developers to avoid the trap of treating a prototype as the first iteration of the project instead of throwaway code. The problem with this approach is that programmers will code beyond the absolute minimum for a prototype if they feel the code is going to evolve into the product. One suggestion to prevent this from happening is to develop the prototype in a different environment or language than production code.

Collaborative design. This practice can range from just informally walking to a coworkers desk and soliciting his or her opinion to scheduling a format inspection. The main point is that some level of review is conducted that is appropriate the level of the project. No one should complete the construction phase without having some feedback.

Build Abstract Data Types (ADT). Creating a collection of data and operations that use that data is an efficient way to encapsulate and hide information to simplify a design. Some of the benefits from using ADTs are [MCCONNELL2004 pg. 127-129)

- Hiding of details

- Changes don't effect the whole program
- The interface can be more informative
- Easier to improve performance
- Program is more obviously correct
- Program becomes self-documenting
- Data is not passed all over the program
- Model real world entities better than low-level structures

Create good interfaces. By applying good abstraction methods and good encapsulation methods, a class interface can provide an abstraction. An interface should adhere to good containments implementation ("has a" relationships) and should have no more than five to nine data members. [MILLER1956]

Follow general member functions and data guidelines. [MCCONNELL2004 pg. 150]

- Keep the number of routines in a class as few as possible
- Don't allow implicitly generated member functions not desired to be public. Use private to keep others from accessing it.
- Minimize the number of different routines called by a class
- Minimize indirect routine calls to other classes
- Minimize the extent to which ha class collaborates with other classes

8.3 Design Techniques

Many techniques that help communicate design idea have been mentioned under general techniques because these ideas are useful for more than just the design phase.

The UML notation is design centric and includes class diagrams and class descriptions in its structural model. Module specifications related to DFDs describe each component to be constructed.

8.4 Case Studies

The original design of the CTAS system became too complex due to the multitude of additional functionality added to the system. This design was developed with functional decomposition. For example, one module handles addition and deletion of flight plans, another interacts with the ISM, and another maintains assignment of aircraft to RAs with similar logic duplicated. The control flow is implicit for system administrators to configure the CM. The main processing loop runs when called by Motif. Here is a partial list of problems addressed by redesign.

- Blocking sends - blocking primitives used to send messages could result in deadlock. Since CM would generate more messages than algorithms could handle, the initial team batched the messages into groups of messages of limited size, causing increased complexity and hard to analyze.
- Failures – CM is a single point of failure. FAA specs indicate that no system outage should be longer than 25 seconds but it takes longer than to reboot the system and refill the aircraft database
- Monitoring – additional monitoring features
- Complexity – system became too complicated

In the redesign, the focus was on reducing the CM's complexity, hoping it would solve other issues. It did this by addressed blocking and monitoring explicitly.

The new design separated parts that cannot be easily reconstructed on reboot from others, with the thought that these would be stored in a persistent database. The redesign team was surprised to find the design could be simplified so much by standard and well-know techniques. They used data abstraction to changed procedure-oriented design into abstract data types that encapsulates data structures and prevent direct access. Infinite queues were changed to use standard message queue abstraction. Deadlocks were avoided deadlocks with the illusion of infinite queue with non-blocking reads/writes. The resulting system reduced the lines of code to about 20% of the original, but only included the core functionality so this metric is of limited use. The lessons learned were:

- simple designs are possible
- standard software engineering techniques work (only used data abstraction, did not need to do object modeling). Before refining processing, make sure the potential of well-understood software engineering notions have been applied
- coding standards are essential – made reengineering the code easier. (information transparency – code written with analysis in mind)
- reverse engineering tools work (helped a lot in rapidly directing to relevant parts of code)
- high level models essential. System level model documentation aids new developers understand system. Inconsistent system models might not be detected until integration

Summary: underlines that the power of software engineering fundamentals such as data abstraction, consistent coding style and design focus on simplicity

8.5 Application to the Small Development Team

Arriving at the appropriate level of design for a project is a difficult process and therefore the small development team must be willing to tolerance some experimentation as it evolves from a more undisciplined approach to embracing modern software engineering practices and principles. Hopefully small teams can be flexible and adjust quickly to those practices that are not working for their projects. Any step toward improving the design process can be helpful. This author recommends that a senior developer of the team work through the suggestions in this paper and generate a well thought out design approach that will work for his or her team. Then he or she should meet with the team and go through a design review for a current project. Any faults in the design or the design methodology can be adjusted until the team generates a working approach. This way the team has ownership in the process and will have a better chance of carrying out the improved process. Management should encourage this exercise, understanding that some benefits of the changes will be immediate but other might take several months to realize such as reuse.

CHAPTER 9

CONSTRUCTION PHASE

The construction phase, also referred to at the coding or implementation phase, is the set of activities including translating the design algorithms into a compliable, programming language that can be executed by a computer. The end result of the construction phase is a program listing (preferably documented) and set of executables. Software construction can be the most expensive part of a project. If requirements definition and design activities are "skimped", then construction will most assuredly be costly, causing this phase to be repeated unnecessarily. In order for this phase to be executed efficiently, the upstream activities must be adequately completed first. The goal of these initial phases is to reduce risk. Therefore it is highly advised not to start implementation without proper preparation.

Steve McConnell lists several reasons why software construction is important in his book Code Complete [MCCONNELL2004 pg. 7] These reasons underscore that efficient construction principles and techniques should be embraced by all developers, regardless how abbreviated the other phases become.

- Construction typically takes from 30% to 80% of the software development project

- Construction is the central software development activity. Requirements and design are performed prior to this phase so that the code can be implemented successfully.
- The source code is often the only accurate description of the software. Documentations from the other phases can be out of date but the code must be current.
- Construction is the only activity guaranteed to get done. No code, no product.

Critical to the success of a small software development team is a well maintained and implemented coding standards document. This document should detail naming conventions of classes, methods, objects, variables, etc. as well as defining acceptable comments and code organization. It is a good idea to establish a required indention convention to provide for consistent readability of the code. For example, having the convention that one and only one class should be written in a single file would allow for easier location and distribution of a class. Having clean conventions established allows new members of the development team to adapt quickly and to resume responsibility for code implemented by other developers. Good conventions should be intuitive and flexible. Since there are multiple equally good and acceptable ways to code, a team should agree upon a single standard. This should not be viewed as limiting a developer's creativity.

Often the development environment for a small development team is established dictated by management. Team members should be trained on how to use the development environment efficiently and uniformly.

Due to the limited number of human resources available for the small development teams, most projects will be written in the languages already mastered by the members of the development team. However the primary criteria on what language to use should be the appropriateness of the language to projects. If none of the languages mastered by the development team are appropriate for the project, then either graciously turn down the project or allow time for retraining of the team members. Large companies often have the luxury of determining the optimal language and then acquiring the team member that are skilled in that language. However with a small development company, having a good generalized "base" coding language is a good approach and the team should either accept projects that can efficiently be implemented in the base language or should requirement management to allow sufficient time for retraining. Obviously, as technology changes, the "base" language used by a team needs to be evaluated to determine if the language is still an efficient way to implement the projects. Retraining a team for a different language can be costly and time consuming practice so the establishment of a good "base" language for the team is critical to reduce the frequency of change.

9.1 Construction Principles

Some general coding principles that should be followed to produce quality code: 1) avoid global variables, 2) write code to read from top down, 3) avoid side effects, 3) use meaningful names for variables, procedures, methods, classes, etc., 4) code for correct functionality before optimizing for speed, 5) modify documentation along with the code , 6) require sufficient requirement and design documentation before coding.

Other principles related to the construction phase that need more discussion are as follows.

Document before coding A good practice is to write the inline documentation of a method before actually coding, then compiling to make sure that the documentation text did not introduce some errors. After a successful compilation, then write the code as specified by the documentation. It is not necessary to provide a line of documentation for each line of code. Each program segment should be commented as to describe its intent. The code should be written in an understandable manner so it is not necessary to describe the detailed logic that is implement but rather the function the segment provides. [DAVIS95 pg. 111]

Hand-execute each component – Each component should be "hand-executed" in an isolated environment with a few simple test cases. This practice could save days of troubleshooting later. [DAVIS95 pg. 112] MicroSoft has provided NUnit testing which allows a small development team to implement this principle with little overhead cost.

Encourage code inspection and peer review - While inspections can take up to 15 percent of development resources, studies show that code inspections can catch 82 percent of coding errors which reduces development cost by 25 to 30 percent with a 50 to 90 percent reduction in testing. [FAGAN]

Avoid tricks – Software developers are often highly intelligent and enjoy showing how clever they are by using "tricks" or coding in obscure ways, like using the

side effort of a function to implement a primary function. The use of tricks may produce the desired results but will not be easily maintained. [DAVIS95 pg. 102]

Avoid deep nesting - Nested IF-THEN-ELSE statements are useful for simplifying code but not more than three levels deep. Nesting more than three levels causing confusing. Some alternatives to deep nesting are to simplify the control structure or to break part of the logic off as a routine. [MCCONNELL pg. 385]

9.2 Construction Practices

The following practices can assist in integrated changes to the system requirements that are presented during the construction phase.

Verify the quality of the new requirement before implementing. Steve McConnell provides a good checklist in his book Code Complete on evaluating the quality of a requirement. [MCCONNELL2004 pg. 42-43]

Communicate the cost of implementing the change. Often the customer will reduce the scope of a requirement or withdraw it once they are aware of the impact on the system and the cost of implementing the new change.

Establish a change control procedure Development teams should allow changes at specified interval to avoid continuous interrupts in the development.

Use development approach that accommodates change. Adopting a phased or prototype process model allows for feedback in an organized, intentional manner.

Be willing to cancel the project. If the requirements are especially bad or too many changes are required, be willing to redo the project or cancel it. The management

should have some concept of how bad a project has to get before cancelling it if nothing more than to provide a comparison to the current state of the project.

9.3 Application to the Small Development Team

The activities of the construction phase are the most common of any developer since coding is the heart of the development process. It is a good place to start implementing more stricture adherence to software engineering best practices and principles for teams that are slow to embrace change. A developer with any experience should be able to recognize their benefits.

Each developer much less each development team has his or her own style of coding. The implementation of these principles and practices can be perceived as hindering the developer's creativity. In order to counter this objection, the management and technical leaders of a company should encourage experienced developers to assist in the development or alteration of the team norms such as coding conventions and provide adequate education on the benefits of proposed changes. The developer that will not conform to the team standards will have to be addressed by the management. Larger companies can reassign a problem development, but in a small company, one member can significantly reduce the benefits of implementing these principles and practices. Having team norms are only useful if the practices actually become the norm. Therefore sometimes management must make the painful decision to release an experienced developer for the long term benefit of team unity.

CHAPTER 10

VALIDATION PHASE

Validation of a software product, or testing, does not stand alone but it is a separate process activity, “the final arbiter of validity before the user assesses its merit” [CHILLAREGE] This phase of the development cycle is the set of activities that performs testing on individual components (unit testing), on sets of unit-tested components (integration testing), and on the entirely integrated set of software components (system testing). The purpose of the tests is to conclude that each component and the system behave as designed. Also the generation of test plans is included in this phase for each of the testing levels. It is important to note that testing can only determine the presence of a bug but cannot determine if software is bug free. Therefore it is important to generate thorough test plans to attempt cover all possible scenarios. Another part of the testing phase is the creation of test harnesses and test environments. One of the challenges of creating a test environment is to replicate multiple clients’ production environment. With today’s vast variety of hardware, operating systems and databases, it is difficult for a small development company to maintain a sufficient enough variety of test environments to cover the environments. Therefore, it is important to discover the environment used by the majority of the perspective clients and decide early on what environments will be supported.

Validation throughout the project is necessary to help ensure that

1) Errors are detected and corrected as early as possible in the software life cycle

2) Project risk, cost and schedule effects are lessened

3) Software quality and reliability are enhanced

4) Management visibility into the software process is improved

5) Proposed changes and their consequences can be quickly assessed

[IEEE 1012 foreword]

Testing should be done throughout all phases of development. While testing activities are documented as a separate phase, the development team should adopt appropriate validation activities throughout the development lifecycle.

- Programmers should use test driven development techniques to define and test modules as they are coding such as MicroSoft's NUnit approach.
- Test plans should be developed concurrently the design and coding phases so that tests are ready to be applied after an incremental release is available.
- Designers should review test cases to see if all functionality is addressed.

10.1 Testing Principles

Tracing tests to requirements. To implement this principle, the requirements must be uniquely identified and listed with an unambiguous description as mentioned in the requirements section.

Plan tests early in the process. Since testing is a major task and critical for program, it should be planned and implemented in parallel with the development

efforts. SRS should be reviewed by the testing team to determine is the requirements are documented in a testable manner prior to the final approval of requirements.

Don't leave testing solely to the developer. While each developer should write tests for test driven code, each component or unit should be tested to some degree by someone else on the team other than the developer who wrote the code. Components should be tested by someone else other than the developer who wrote the code. Also test plans should be written by someone else other than the developer who wrote the code and then reviewed by the developer to ensure tests are appropriate.

Don't only test the "happy path". Successful tests find faults in the expected use of the component but also undesirable results when the component is "misused". A complete test plan should include testing invalid inputs and should stress test a system. If there are constraints the use or environment of a system, then these should be adequately documented.

Testing needs to be complete. The development team needs to be committed to not skipping or short cut testing, shipping until tests are completed or integrating until unit test passes. While this seems obvious, it is a strong temptation to not complete testing due to deadline pressure or desire to move on to the next project.

10.2 Testing Techniques

Use a requirements/test table. A tangible way to verify if all tests are linked to a requirement is to create and maintain a large binary table with the rows representing the tests and the columns representing each requirement. Then each corresponding cell is flagged with a 1 for the intersection of a test and a requirement.

This table provides a simple approach to determining is all requirements are captures by at least one test and if all tests are based on at least one requirement.

Use state task diagram. The diagrams visualize transitions from one state to another and should be used to generate test case by validating the claims and completeness of the state task diagram.

Use scenarios to test. If using the UML notation, then the development team need to commit to writing good scenarios which capture all the requirements.

10.3 Testing Practices

Combine "Black box" and "White box" testing "Black box" testing only uses the external behavior description as its criteria for testing. Input and output expectations are limited to what was documented that this component should or should not do. "White box" testing is based on the actual logic within the code. to generate test. Given these definitions, it stands to reason that developers should lean toward white box testing and complete documentation of each component's external behavior modules while the test team do only black box testing, assuming that the testing team members are not skilled in reading code. Both modes of testing are necessary and should complement one another.

Work from functional specifications. Functional specification should have been created in some form during the requirements phase for the purpose of not only describing the system but providing harmony between testing and design. Testers use this document to write test cases. This practice can be executed in parallel with development.

Develop automated test execution. The goal of automating test execution is to reduce the hours of manual work involved in test execution. Small development teams should consider investing into some test automation tools. Once integrated in to a team's mode of operation, then the results should be shorter and more consistent testing phases.

Utilize user scenarios. User scenarios were introduced early in this paper to define requirements but can also be used for testing. Scenarios have the advantage of testing the functionality from the end user's perspective. Therefore it is helpful for the test team member to be involved with writing the user case studies to adequately reflect the user experience.

Team test with developers. Close coupling of testers with developers improves the test cases and the code developed. One practical method of executing this practice is to have testers write test cases and then walk through the test with the developer that will be coding the functionality. Microsoft goes so far as to have the tester shadow the developer, which is not cost justifiable for the small development team but underscores the importance of this practice. The practice can serve as guidance during the team selection or even the hiring process.

Test for memory leaks. Memory resource failures can be simulated to some extent with commercial tools. Memory leaks are caused by poor resource handling in the source code or in third party APIs.

CHAPTER 11

INTEGRATION AND DEPLOYMENT PHASE

"Integration assembles the application from the set of components previously implemented and tested. Deployment is the handing over of a system to customers for production use." [MACIASZEK pg. 19] Deployment is the most important phase for the software development company since it is the phase that generates income. Most of the company is not interested in the development problems but just want to know that it works and can be delivered to the clients. This realization is overlooked some times but developers who are closely attached to the development lifecycle. A good practice is to focus on the end and adjust accordingly within the principles mentioned throughout this paper.

Deployment is also the phase with potentially the least amount of control. Every client's environment will have its own uniqueness and potential to encounter new problems. The company's installation specialist should evaluate the clients IT environment and determine what environments (OS, database version, third party application, etc.) will be supported for this release of the product. [BIRTLEY]

Deployment is more than installation. Installation is the act of loading artifacts onto a host environment. Deployment is doing whatever is required to get the application into production use on the client's environment so that the product will start

generating income for the development company. It certainly involves generating the installation procedures and disks, delivering the training material, troubleshooting integration issues.

The author's experience with deployment is that this task has been assigned to other IT staff besides the developers. The activities in this phase include the creation of installation images, production of user documentation, installation of product, loading of databases, training of users and reporting of problems during integration at the client site. Often it is efficient to have these same staff in the small development company be the testers as well as product support. In this environment, the "programmers" would provide the support staff the executable components with the necessary support documents on how to operate the system and move on to the next project.

With this approach the development team must be expanded to include the staff with this skill set. With such an environment, the installation and training staff can specialize at their tasks but would be dependent on the developers to generate the appropriate documentation. The installation specialists should be provided opportunity to provide feedback during the beginning of the design phase and during requirements review, especially for the redesign of projects that they have been supporting.

This phase has its own set of issues to be addressed that are important to the success of the project but most are outside of the scope of this paper. One of the activities of this phase is the build process. "The build" is more than just compiling In the XP process mentioned early in the paper, one of the practices is to automate the build process to be completed in less than 10 minutes and to execute this build at least

once a day. While a development team might not embrace all of the XP methodology, this practice seems reasonable to be incorporated into every project, regardless of the process model chosen. If this practice is implemented from the start, then the staff responsible for actually deploying the system can implement some realistic mock deployment tests, and can determine early one if the manner in which the build is produced is conducive to being deployed. If issues are found, then the team has time to respond before the release of the product.

CHAPTER 12

MAINTENANCE PHASE

Maintenance is the phase that follows a successful hand over to a customer of each incremental release, eventually the entire software product. This phase is an inherent part of the software cycle and has been said to account for 67% of the entire software life cycle. [SCHACH] "Maintenance typically consumes 40 to 80 percent of software costs. Therefore, it is probably the most important life cycle phase of software." [GLASS pg. 115] Given this activity traditionally accounts for such a high percentage of software development, it stands to reason that sufficient efforts should be made to improve maintenance activities. However, some argue that reducing the maintenance cycle is an unobtainable goal. Better software engineering development has lead to longer maintenance phases because changes are easier to make, thus resulting in more code changes. Better software also lasts longer and therefore has a longer maintenance life. Therefore a better metric for determining improvement in this phase might be the turn around time for bug resolution or enhancement incorporation, rather than how many man-hours are spent on the maintenance of a project.

Often the original developer will not be the one that will maintain the system, requiring others to learn the system well enough to make efficient alternations. Understanding the existing product can take 30 percent of the total maintenance time

and is the dominant maintenance activity. [GLASS pg. 120] If for not other reason, this fact validates the need for good documentation and archival system.

The maintenance phase has three stages : Corrective (addressing faults), Adaptive (adjusting functionality to satisfy the changing environment), Perfective (modification to accommodate new or significantly altered requirements). [MACIASEK90] Each of these stages have distinctive goals and is helpful to break down the maintenance activities for discussion.

12.1 Corrective Stage

While bug fixing will always be part of the maintenance, proper adherence to the principles should greatly reduce the time required for this activity. Often the initial fix is temporary, just enough to keep the system going. The problem should be reported and properly address in a future release.

12.2 Adaptive Stage

Activities in this category are unavoidable for keeping software usable. This stage does not result in new capabilities for the user but is limited to only the activities performed to change the system to execute properly in an altered environment. Events like changes to the operating system, hardware platform, compiler, software library or database structure would lead to adaptive maintenance activities. These events are driven by the market and clients desire to keep current with their computing environment.

12.3 Perfective

This stage is the most unpredictable and time consuming but can be the most controllable. Activities in this stage are those that implement new requirements or an alteration of existing requirements. Obviously new requirements cannot be predicted but management does have the control over whether these enhancements will be implemented. "Enhancement (perfective stage) is responsible for roughly 60 percent of software maintenance costs. Error correction (adaptive phase) is roughly 17 percent. Therefore, software maintenance is largely about adding new capability to old software, not fixing it. " [GLASS pg. 117] A small development team should adapt a systematic procedure of change control, from the manner in which the enhancement is defined to how it is implemented and tested. Keeping this stage in mind while designing a new system can save countless hours later when new requirements are to be implemented. Some authors, such as Pfleeger, add a preventive maintenance phase that focusing on fault handling to make sure the system can handle all possible conditions.

CHAPTER 13

CONCLUSION

This paper has presented the software engineering best practices and principles for software development discovered by this author. In this chapter, the author will provide some concrete applications of this material to the software development team with fewer than eight members. Several scenarios are presented with the author's suggestions.

Throughout the research for this paper, four reoccurring themes for improving the efficiency of a small development team were

- to generate traceable requirements
- to seek feedback as early in the process as possible
- to incrementally develop the product
- to plan for reuse

These activities provide a good foundation for the improvement of any development team and foster many other principles and practices discussed. The implementation of these practices take a conscious effort but it is the opinion of this author based on his research that this effort will rewarded significantly. Traceable requirements enable requirements, design and test activities to be directly linked to one another. Incremental releases enable constructive early feedback. Early feedback encourages reviews of various types. Planning for reuse drives modular designs, coding

standards and test driven development because the developer knows that others will use and depend on his or her code.

13.1 Development Environment

The management and technical leadership of a small development company should work together to establish technical norms of software development. A strong technical leader with good communication skills should be endowed with the responsibility of encouraging and enforcing these norms. Without this role, then the developers will follow their own preferences and therefore any sense of standardization will be lost. Team norms without enforcement are just suggestions. It takes a skilled, creative and tactful person to enforce norms without limiting creativity and job satisfaction. Norms should include universal best practices regardless of the size of the teams. In addition to actively investing in training to equip the development team, the company should provide an adequate environment to enable the developers to be successful as software engineers. This environment includes

- modern Integrated Development Environment such has MicroSoft Visual Studio.NET
- software configuration management tools like Seapine SurroundSCM
- modeling tools like MicroSoft Visio
- stable network with reliable backup system,
- knowledgeable and available technical support
- adequate desktop equipment (large enough monitors, enough RAM and CPU horse power)

- automated testing tools

As key employees evolve in their understanding for the company's core products and approach to software development, management should transition these employees into technical leadership positions (not necessarily management roles) and provide them the resources to mentor the development staff in the company's development norms. These norms should always include

- always be conscious of requirements, design and software component reuse
- requirements reviews
- technical design reviews
- requirements lists with unambiguous reference and unique id
- acceptable design methodologies appropriate to the project
- coding conventions
- configuration management conventions (how to store source code and related documentations and how to implement version control)
- testing responsibilities and expectations clearly defined

Granted that for smaller projects, many of these activities can be combined and greatly shortened but some version should be executed in order to enforce the company's standards and generate consistent, stable products. Regardless the size of the team, each company should have senior technical staff monitoring the norms and conducting technical reviews. This resource can also be directly involved with the requirement definition, design and testing phases of several projects.

13.2 Scenario One – The Lone Developer

Often, a small project is assigned to a single developer to execute most all of the activities of the software lifecycle. This developer should be familiar with all aspects of software development and should consult with his management on what level of documentation is expected. At a minimum, the developer should generate a "high level" requirements list to assist in communicating the requirements in the staff responsible for integration testing. The developer should request for a testing "partner" to assist in writing the test cases and conducting the tests in order to implement the practice of not allowing the developer to completing test his or her own code.

Periodic informal reviews should be conducted by another "lone developer" or by senior technical staff. If possible, the developer should use the incremental process model with two week incremental or iterative release cycles. The management can act as the customer if it is not practical for the customer to review and test the release each two weeks. This practice is particularly important for the lone developer to prevent costly misunderstandings from escalating over months.

.It is tempting to neglect these best practices and principles for small projects because the resulting software can in some case be produced more quickly without taking time.for the "extras". However, this omission will most likely lead to the following undesirable conditions.

1. No duplication of knowledge. Each "lone developer" might have the tendency to develop in isolation. The obvious problem is the inevitable costly learning

curve required for another developer to pick up the project if that developer is unavailable or leaves the company and a change is required.

2. Inadequate design documentation. Since the developer is doing the design and construction himself, then he might be tempted to not complete the design documentation and therefore leave run for assumptions to go unchecked during design reviews.

3. Progress is dependent on one individual. If the developer is out of the office or is temporarily assigned to another project, the progress on his or her project stops. If the delay to return to the project is significant (highly variable on the developer's memory retention), then significant time can be spent for the developer to get "back in the flow" of developing this project.

4. Maintenance is complicated if only one developer is knowable about the program and little to no documentation exists. Extra time is required to fix a bug or add an enhancement even for the original developer if enough time has elapsed since its release.

For the company with several "lone developers", task swapping mentioned earlier in this paper is a concept that might be worth considering overcoming some of the negatives if pair programming is not an option. The "partner" can keep the project going during the developer's absence and provide the extra accountability.

13.3 Scenario Two – The Tiny Team

Several small companies may employ two to three developers for a project. This approach has much promise for the small development company as it directly

addresses the negatives of "lone developer" environment. With the "tiny team" approach, many of the advantages of the XP method can apply, regardless if XP is formally embraced or a more traditional approach is followed.

It is important that each team member recognize his or her role. While each team member can contribute and review requirements and designs, one person should be chiefly responsible for documenting and communicating the results of these phases. The teams should work from the design to avoid duplicating effort and to ensure that components follow the specified interfaces. Each team member should be aware of the strengths of others and seek to learn from them. If skills allow, then roles should be swamped when transitioning to another projects. Coding standards, test driven development, team design reviews can be used as unifying events for the small team.

13.4 Convincing Management

One of the challenges for the development team to face when transitioning into efficient "programming in the medium" approach is to convince management to support this cultural change. When management is not convinced that construction prerequisites are worth the effort, then the software engineer is faced with a dilemma. Steve McConnell lists the options for developers in this predicament. If the manager requires a developer to start coding right away, then the developers' choices are : 1) give in and just do it to satisfy the management, 2) pretend to coded and proceed with requirements and design, 3) educate the management on the advantages of "doing it right", or 4) find a new job in a more enlightened company. [MCCONNELL2004 pp. . 26-27]

13.5 For All Small Teams

In summary, all small development teams should consider the following practices. These have been proven in other environments to be helpful, radically in some cases.

- Nightly 10 minute builds
- Reuse
- 10 minute build
- Early feedback
- Incremental releases

While it was not possible to cover all of that modern software engineering has to offer the small development team, hopefully some of the concepts and ideas presented in this paper have reminded and encouraged developers who find themselves transitioning to "programming in the medium" to become more efficient though applying the collective wisdom of the industry experts.

REFERENCES

Alhir, Sinan Si (1998) UML in a Nutshell, O'Reilly

Beck, Kent & Andres, Cynthia (2004) Extreme Programming Explained: Embrace Change 2nd Edition, Addison-Wesley Professional

Birtley, John. (2005) Best Practices for Risk-Free Deployment http://www.theserverside.com/articles/article.tss?l=BuildManagement (accessed last on March 10, 2006)

Boehm, B. (1976) "Software Engineering", IEEE Transactions on Computers, 25, 12 (December 1976), pp. 1226-1241

Chillarege, Ram. (April 26, 1999) Software Testing Best Practices, IBM Research, http://www.chillarege.com/authwork/papers1990s/TestingBestPractice.pdf (accessed on March 4, 2006)

Cunningham, Ward. (2001) "Manifesto for Agile Software Development" (2001) http://www.agilemanifesto.org (accessed on March 4, 2006)

Davis, Alan E. (1995) 201 Principles of Software Development. McGraw-Hill, Inc.

Factguru. Software Development Team http://www.site.uottawa.ca:4321/oose/index.html#smallsoftwaredevelopmentteam

Fagan, Michael "Design and Code Inspections to Reduce Errors in Program Development", IBM System Journal, 15, 3, July 1976

Fichman, Robert M. "Incentive Compatibility and Systematic Software Reuse", Journal of Systems and Software, NY. Apr 27, 2001, Vol. 57, Iss. 1, pg 45

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John, (2002) Design Patterns, Elements of Reusuable Object-Oriented Software. Addison- Wesley.

Glass, Robert E. (2003) Facts and Fallacies of Software Engineering, Addison-Wesley

Humphrey, Watts (2006) "Sweet Predictability", Software Development, February 2006, pp. 35

IEEE Std 1002-1987. IEEE Standard Taxonomy of Software Engineering Standards (June 4, 1987)

IEEE Std 1012-1986. IEEE Standard for Software Verification and Validation Plans (September 17, 1986)

IEEE Std 1016-1998. IEEE Recommended Practice for Software Design Descriptions (September 23, 1998)

IEEE Std 1028-1997. IEEE Standard for Software Reviews (December 9, 1997)

IEEE Std 1063-2001. IEEE Standard for Software User Documentation (December 20, 2001)

IEEE Std 1074-1997. IEEE Standard for Developing Software Life Cycle Process (December 9, 1997)

IEEE Std 1219-1997. IEEE Standard for Software Maintenance (December 9, 1997)

IEEE Std 1420.1-1995. IEEE Standard for Information Technology – Software Reuse Data Model for Reuse Library Interoperability: Basic Interoperability Data Model (BIDM) (December 12, 1995)

IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. (September 28, 1990)

IEEE Std 829-1998. IEEE Standard for Software Test Documentation. (September 16,, 1998)

IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications. (June 25, 1998)

Jackson, Daniel & Chapin, John "Redesigning Air Traffic Control", MIT Lab for Computer Science, May/June 2000, IEEE Software, http://people.csail.mit.edu/dnj/publications/ctas00.pdf

Kulak, Daryl & Guiney, Eamonn (2000) Use Cases Requirements in Context, Addison-Wesley

Lai, Robert Chi Tau Process Definition and Modeling Methods. Technical Report SPC-91084-N Hendon, VAP: Software Productivity Consortium.

Lehman, M.(1991) "Software Engineering, the Software Process and Their Support", Software Engineering Journal, 6, 5 (September 1991, pp. 243-258, Section 3.6

Lutowski, Rick (2005) Software Requirements Encapsulation, Quality, and Reuse. Auerbach Publications.

Maciaszek, Leszek A., Bruc Lee Liong (2005). Practical Software Engineering: A Case Study Approach. Pearson Education Limited.

Magee, Stan & Tripp, Leonard L. (1997) Guide to Software Engineering Standards and Specifications, Artech House

Magee, Stan.(2006) "Software Engineering Standards Suite Selection". http://www.12207.com/std.%20Suite.htm (last access March 2006)

McCabe, Tom, "A Complexity Measure", IEEE Transactions on Software Engineering, 2, 12, December 1976, pp. 308-320.

McConnell, Steve, (2004) Code Complete. 2nd Edition. MicroSoft Press

Miller, G. (1956) The magical number sever, plus or minus two. Some limit on our capacity for processing information. Psychology Review 63:2

Miller, Ann, Pressman, Roger & Yourdon, Edward "16 Critical Software Practices – Inspect Requirements and Design", www.iceincusa.com/16CSP/content/14_inspt/insrgt.htm

Mills, H. "Top-Down Programming in Large Systems", in *Debugging Techniques in Large Systems*, R. Ruskin, ed., Englewood Cliffs, N.J.: Prentice Hall, 1971

Moore, James W. (1998) Software Engineering Standards – A User's Road Map. IEEE Computer Society Press

Parnas. D.L. & Clements, P.C. (1986) "A rational design process: How and why to fake it" IEEE Transactions on Software Engineering SE-12.2 (February) pg. 251-257.

Pfleeger, Shari Lawrence, (1998). Software Engineering Theory and Practice. Prentice Hall

Pressman, Roger S. (2001) Software Engineering: A Practioneer's Approach, 5th edition, McGraw-Hill series in computer science

Royce, W. W., (August 1970) "Managing the development of large software system: Concepts and Technique." Proceedings of WESCON

Robertson, James & Robertson, Suzanne, (1994) "Complete Systems Analysis: The Workbook, the Textbook, the Answers" Dorset House Publishing

Schach, S. (1996) Classical and Object-Oriented Software Engineering. 3rd edition, Irwin, pp. 604

The Standish Group Report, "Chaos", http://www.projectsmart.co.uk/docs/chaos_report.pdf (last accessed March 3, 2006)

SESC Long Range Planning Group, "Master Plan for Software Engineering Standards", Version 1, Dec 1, 1993

Wikipedia. The Free Encyclopedia (2004) "Coupling and cohesion". Web page and wiki,. http://c2.com/cgi/wiki?CouplingAndCohesion (accessed on March 4, 2006)

Wikipedia. The Free Encyclopedia "Agile software development", http://en.wikipedia.org/wiki/Agile_software_development (accessed on March 4, 2006)

Wikipedia. The Free Encyclopedia “Methodology”, http://en.wikipedia.org/wiki/methodology (accessed on March 4, 2006)

Wikipedia. The Free Encyclopedia “Practice”, http://en.wikipedia.org/wiki/practice (accessed on March 4, 2006)

Wolber, David, CS 112: Supplementary Notes Structure Charts and Bottom-up Implementation, http://www.usfca.edu/~wolberd/cs112/SupplementalNotes/structureChart.doc

Zimmer, J A. (1996) Programming In the Large Versus Programming In the Small http://www.mapfree.com/sbf/tips/smalarg.html, (accessed on March 4, 2006)

BIOGRAPHICAL INFORMATION

Miles Phillips received his Bachelor of Science in Computer Science from Baylor University in Waco, TX, in 1986. Since then, he has worked for the following companies: E-Systems (defense contractor), North Central Texas Council of Governments (local government planning), Berger and Company (IT consultant), Union Pacific Resources (oil and gas exploration), University of Texas at Arlington School of Urban Affairs (higher education), and LandWorks, Inc. (commercial GIS software development). During the past five years, he has been working as a software developer from his home for LandWorks, Inc, which is based in Houston, TX. He primarily designs and builds Geographic Information Systems (GIS) applications using Oracle PL/SQL, MicroSoft SQLServer TSQL and with ESRI's ArcMap Suite of products. Mr. Phillips has also held positions as a GIS consultant, Oracle database administrator, UNIX system administrator, GIS project manager, GIS adjunct professor and IT manager.

He plans on continuing his career as a software engineer with an emphasis on database and GIS applications and would like to teach computer science at the university level in the near future.

Mr. Phillips is married to Kristi and they have four children Emily, Elizabeth, Ellen and Benjamin. They currently live in Aledo, TX.