

California State University, San Bernardino

CSUSB ScholarWorks

Theses Digitization Project

John M. Pfau Library

1997

The recursive multi-threaded software life-cycle

Scott James Simon

Follow this and additional works at: <https://scholarworks.lib.csusb.edu/etd-project>



Part of the [Software Engineering Commons](#)

Recommended Citation

Simon, Scott James, "The recursive multi-threaded software life-cycle" (1997). *Theses Digitization Project*. 1306.

<https://scholarworks.lib.csusb.edu/etd-project/1306>

This Thesis is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.

THE RECURSIVE MULTI-THREADED
SOFTWARE LIFE-CYCLE

A Thesis
Presented to the
Faculty of
California State University,
San Bernardino

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Scott James Simon
June 1997


THE RECURSIVE MULTI-THREADED
SOFTWARE LIFE-CYCLE

A Thesis
Presented to the
Faculty of
California State University,
San Bernardino

by
Scott James Simon


June 1997

Approved by:

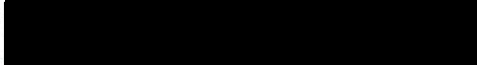


Dr. Arturo I. Concepcion, Chair, Computer Science

02 Jun 97
Date



Dr. Richard J. Botting



Dr. Kerstin Voigt

© Copyright 1997

Scott J. Simon

Abstract

Software life-cycles are aimed at improving the process of developing software. Traditional life-cycles are adequate for structured development, but not for object-oriented software. Object-oriented software development has a different “style” than structured methods which requires different considerations by a software life-cycle. There are a number of existing object-oriented life-cycles that address the specific needs of object-oriented development, but these have little or no support for monitoring progress during development and contain limitations.

This thesis presents the recursive multi-threaded (RMT) software life-cycle which supports the monitoring of progress during development, addresses the specific needs of developing object-oriented software, and attempts to resolve deficiencies found in existing life-cycles. RMT uses the logical concept of a “thread” for partitioning and organizing development activities during the development process, which makes it unique from existing life-cycles. Threads support iteration and recursion which will be shown to be critical concepts for object-oriented development. The use of threads also provides a mechanism for measuring progress, provides a hierarchical structure for organizing team members, clearly delineates responsibilities, and identifies well-known paths of communication among team members.

First, the motivation and requirements for RMT are defined, followed by a brief summary of a number existing software life-cycles illustrating their limitations by comparing them to the previously defined life-cycle requirements. Next, the components of RMT are defined in addition to an example of applying RMT to a sample project. In

conclusion, the strengths/weaknesses of RMT, RMT's relevance to the Capability Maturity Model (CMM), and future directions of RMT are also discussed.

Acknowledgments

This thesis represents a major milestone in my life. Even though it has only taken three years to complete, it represents a significant expenditure of time, effort, and commitment (and not just by myself). It is more than simply a collection of words recorded on paper, it represents the completion of a substantial goal that I set more than three years ago. While the research portion of this thesis was what I expected it would be, the writing of the thesis was nothing like what I anticipated. Being my first undertaking in writing a major paper of my own ideas, I realized the difficulties and frustrations a writer faces when trying to record the thoughts and ideas flying around their head into a computer (☺). Professional writers, whether they author magazine articles, technical articles, or books, have earned my respect.

This thesis is more than simply “my” work. While I was the author, it is the result of input and support from many. First, I would like to thank the Associated Students, Incorporated (ASI) at California State University, San Bernardino who helped support my research through an ASI award. I would also like to acknowledge my committee members Dr. Richard J. Botting and Dr. Kerstin Voigt for all of their insightful comments and suggestions, pointing out not only technical mistakes but also the conceptual flaws during the development of this thesis, making it better than I could have alone.

My thesis advisor Dr. Arturo I. Concepcion deserves credit for much more than “advising” me during this journey. Not only did he give me ideas and suggestions on my thesis, but he also guided me when I was unsure and was supportive when I was in doubt. He has been more than an advisor, he has been a friend.

I would like to thank my mother, for molding me to be the man that I am today. I want to thank her for all of the things that she has given me over the years, from advice, to a strong sense of right and wrong, and for that really cool motorized car for Christmas when I was a child; and I want to thank her for all of the things that she sacrificed to give them to me. Thanks for being the best mother someone could ever hope for. I just hope I can be half of the parent to my son that you are to me.

Finally, and most importantly, I would like to thank my loving wife Renee. She has been so understanding and supportive of me throughout these past three years. She has sacrificed precious time that we could have spent together while I toiled away at the computer or met with my committee members, but she never complained. Rather, she supported me with words of encouragement and telling me how proud she was of me. She has given up as much or more as I have, and deserves as much or more of the credit. Renee, I am proud of you...this is for you.

Table of Contents

| | |
|--|-----|
| Abstract..... | iii |
| Acknowledgments | v |
| List of Tables..... | xi |
| List of Illustrations..... | xii |
| Chapter One-Introduction..... | 1 |
| 1.1 The Recursive Multi-Threaded (RMT) Software Life-Cycle | 3 |
| 1.2 Motivation for an Object-Oriented Software Life-Cycle..... | 4 |
| 1.3 Recursive Multi-Threaded Life-Cycle Requirements | 6 |
| 1.3.1 Traditional Life-Cycle Requirements | 7 |
| 1.3.2 Object-Oriented Life-Cycle Requirements | 9 |
| 1.4 Capability Maturity Model (CMM)..... | 13 |
| 1.5 Structure of Thesis | 14 |
| Chapter Two-Existing Software Life-Cycles | |
| 2.1 Taxonomy of Software Life-Cycles | 15 |
| 2.2 Waterfall Model | 16 |
| 2.3 Spiral Model | 17 |
| 2.4 Round-trip Gestalt Design..... | 18 |
| 2.5 Recursive/Parallel Model..... | 19 |
| 2.6 Fountain Model..... | 20 |
| 2.7 Chaos Model/Life-Cycle..... | 23 |

| | |
|---|----|
| 2.8 McGregor and Sykes..... | 24 |
| 2.9 The Visual Modeling Technique (VMT)..... | 27 |
| 2.10 Methodology for Object-oriented Software Engineering of Systems (MOSES) | 28 |
| 2.11 Common Limitations | 29 |
| Chapter Three-The Recursive Multi-Threaded (RMT) Life-Cycle | |
| 3.1 Process Concepts | 31 |
| 3.1.1 Threads..... | 32 |
| 3.1.1.1 Iterative/Evolutionary Development | 35 |
| 3.1.1.2 Recursion..... | 40 |
| 3.1.1.3 Reusability..... | 43 |
| 3.1.1.3.1 Source Code Reuse..... | 44 |
| 3.1.1.3.2 Design Patterns..... | 46 |
| 3.1.1.3.3 Open-Ended Architectures | 47 |
| 3.1.2 Benefits of Threads..... | 52 |
| 3.1.2.1 Monitoring Progress..... | 52 |
| 3.1.2.2 Multiple Abstraction Levels..... | 54 |
| 3.1.2.3 Parallel Development..... | 57 |
| 3.2 RMT Activities/Phases | 61 |
| ✓ 3.2.1 Requirements Analysis | 62 |
| 3.2.2 Planning..... | 63 |

| | |
|---|----|
| 3.2.3 Analysis | 65 |
| 3.2.4 Design | 67 |
| 3.2.5 Implementation | 69 |
| 3.2.6 Quality Assurance | 70 |
| 3.2.6.1 Risk Management | 72 |
| 3.2.6.1.1 Risk Analysis | 72 |
| 3.2.6.1.2 Risk Monitoring and Avoidance | 75 |
| 3.2.6.1.3 Risk Resolution | 75 |
| 3.2.6.1.4 RMT Risks | 76 |
| 3.2.6.2 Traceability | 78 |
| 3.3 Documentation | 79 |
| Chapter Four-Applying RMT | |
| 4.1 The Project | 80 |
| 4.1.1 Thread Naming Convention | 80 |
| 4.2 The First Iteration | 82 |
| 4.3 The Second Iteration | 87 |
| 4.4 The Third Iteration | 89 |
| 4.5 The Fourth Iteration | 91 |
| 4.6 Additional Considerations | 91 |
| 4.6.1 Iterations and Child Threads | 91 |

| | |
|--|-----|
| 4.6.2 Early Termination of a Thread (Handling Inconsistencies/Defects)..... | 92 |
| 4.6.3 Managing Multiple Abstraction Levels | 93 |
| 4.6.4 Methodologies | 94 |
| Chapter Five-Conclusions | 96 |
| 5.1 Relevance to the Capability Maturity Model (CMM) | 98 |
| 5.2 Future Directions | 100 |
| Appendix A-Glossary..... | 102 |
| Bibliography | 106 |

List of Tables

| | |
|--|-------|
| Table 2.1. Taxonomy of Software Life-Cycles | 16 |
| Table 3.1 Gilb's Basic Principles of Open-Ended Design..... | 49 |
| Table 3.2 Documentation Generated During Thread Phases | 79 |
| Table 5.1 CMM Maturity Levels..... | 98-99 |

List of Illustrations

A. List of Figures

| | |
|---|----|
| Figure 1.1. Conventional vs. Iterative Life-Cycles | 12 |
| Figure 2.1 Spiral Model | 18 |
| Figure 2.2 Fountain Model..... | 22 |
| Figure 2.3 Linear Problem-Solving Loop | 24 |
| Figure 2.4 Fractal Problem-Solving Loop..... | 24 |
| Figure 2.5 McGregor and Sykes Application Life-Cycle..... | 25 |
| Figure 2.6 McGregor and Sykes Fractal Software Development Process..... | 26 |
| Figure 2.7 Overall MOSES Life-Cycle | 29 |
| Figure 3.1 RMT Thread Activities/Phases | 34 |
| Figure 3.2 RMT Thread with N-Iterations..... | 36 |
| Figure 3.3 RMT Thread with Recursion..... | 43 |
| Figure 3.4 Levels of Thread Abstractions and Thread Managers..... | 56 |
| Figure 3.5 Distribution of Activities, in Practice, of Traditional, Sequential Software Life-Cycles..... | 58 |
| Figure 4.1 Thread Hierarchy of the First Iteration | 87 |
| Figure 4.2 Thread Hierarchy of the Third Iteration..... | 90 |

Chapter One-Introduction

Because our society has become heavily dependent on computers, the software those computers execute has been given great responsibilities. Because of this responsibility, the repercussions of software failures can be significant, even resulting in the loss of human life. Between 1985 and 1987 at least two people died of radiation overdoses by the Therac-25 medical linear accelerator as a result of a fault in the control software [Leveson-93]. Also, in the 1991 Gulf War a fault in the software for the Patriot missile caused a Scud missile to penetrate the Patriot anti-missile shield near Dhahran, Saudi Arabia, killing 28 Americans and wounding 98 [Mellor-94]. Why do such significant software failures continue to occur? The answer is simple, human beings make mistakes. It would seem, however, that with all of the advances in software engineering and technology in the past half-century that such critical software systems could be developed with better reliability. It is obvious that this is not the case and process of developing software has room for improvement.

Software is complex, for many reasons. The problems software is intended to solve are complex; the software itself is complex; and coordinating people to build software is complex. Frederick P. Brook's, Jr. pointed out that in order to generate an order of magnitude improvement in the development of software, the essential difficulties of software development need to be addressed, rather than the accidental difficulties [Brooks-95]. These essential difficulties include the inherent complexities found in the nature of software and its development. Accidental difficulties are problems with the production, or realization, of the software with today's technology, which are not inherent

to the software. The essential difficulties include deciding how software is developed and what is developed, not the actual implementation, or coding, or the software.

Software engineering is a discipline whose goals are, simply put, to manage and/or eliminate these essential difficulties of software development to produce better software, make the process of developing software easier, and to do it in a productive fashion. Fritz Bauer provided an early definition of software engineering as “the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.” [Naur-69]

One area of software engineering aimed at improving how software is developed is the definition of a repeatable, systematic process that can be applied to the construction of software, called a software life-cycle. A repeatable process helps eliminate many of the uncertainties common to software development. In order to create a repeatable process, a software life-cycle defines a set of activities, what tasks are performed during each activity, the order that the activities occur, the preconditions that must be met before beginning an activity, and the postconditions that must be met before an activity is complete. Some common activities included in life-cycles are analysis, design, coding, and testing. These activities, and the life cycle itself, are intended to make the development effort more efficient, so it is equally important that the process does not impede the work of the developers.

A life-cycle must address the needs of many people involved in the development process. For software engineers, a life-cycle should provide a step-by-step procedure to follow for developing software. For project managers, a life-cycle should provide

mechanisms for coordinating development activities, monitoring progress, allowing the development staff to communicate effectively, and (most importantly) to generate quality software that satisfies the system requirements.

A software life-cycle is a process, not a methodology. Software methodologies focus on how to approach and solve a particular class of problems, while a life-cycle is a process organizes the steps taken to solve that problem. Methodologies are used within the framework of a life-cycle. Sometimes methodologists define a life-cycle and a methodology together, like MOSES [Henderson-Sellers-94], making the division between the life-cycle and the methodology vague and confusing.

1.1 The Recursive Multi-Threaded (RMT) Software Life-Cycle

The recursive multi-threaded (RMT) life-cycle proposed in this thesis is a software development process which supports the monitoring of progress during development and addresses the specific needs of developing software using object-oriented technology. A number of object-oriented software life-cycles exist today, but they have little or no support for monitoring progress during development, are simply general concepts that lack detail, and/or have other limitations (which will be shown later). RMT is based on many of the same fundamental concepts found in other object-oriented life-cycles, but it is a detailed life-cycle which attempts to resolve limitations found in existing life-cycles. A severe limitation of existing life-cycles that RMT addresses, is the ability to monitor progress during development. What makes RMT unlike existing life-cycles is its use of an abstraction, called a thread, to organize the development process. Two distinguishing characteristics of RMT are iteration and recursion. As will be shown later, iteration is an

inherent trait of successful object-oriented projects and recursion provides developers with an effective technique for organizing the development process, to monitor progress, and to allow efficient communication between team members. This thesis will show the motivation and requirements of RMT, limitations of existing object-oriented life-cycles and how RMT resolves those limitations, and a detailed description of what RMT is and how it can be applied to projects.

1.2 Motivation for an Object-Oriented Software Life-Cycle

Aside from the need for better software development processes because of system failures, there is a need for developing an object-oriented life-cycle that facilitates the monitoring of progress during development. As will be shown in chapter two, existing life-cycles have little or no support for monitoring progress and/or the structure of existing life-cycles makes progress monitoring difficult. The ability to measure progress during development is significant because it allows managers and developers to determine whether a project is on schedule or not. When a project overruns some planned schedule, the ability to monitor progress during development can help identify that the project is behind schedule earlier during development, rather than at the final delivery date, allowing managers/developers to take appropriate actions to accommodate the situation.

Another motivation is that there is a demand for object-oriented life-cycles because traditional life-cycles are ill-suited for object-oriented technology. While the history of object-oriented programming and object-oriented techniques date back to the 1960's, it was not until the 1980's that object-oriented technology began to be widely used within the software engineering community. Prior to the wide spread use of object-oriented

technology, there were a number of software life-cycles based upon “traditional” non-object-oriented technologies. However, object-oriented technology takes a different approach to software development than procedural methods. The object model focuses on entities (objects), their attributes, and their behavior rather than placing the emphasis on functions. Due to this significant difference (and others) between procedural and object-oriented methods, many traditional life-cycles simply do not address the requirements specific to the development of object-oriented software (see chapter 2). Some specific requirements that some traditional life-cycles do not support are iteration or the overlap of development activities, which are common for object-oriented projects.

An illustration of this demand for object-oriented life-cycles is that many individuals and organizations expend significant effort to developing better processes, demonstrating that new processes are needed because existing life-cycles do not meet their needs. As a result of this effort, new life-cycles continue to be developed and published. For example, a team at the IBM ITSO San Jose Center in California began working on a life-cycle and methodology called the Visual Modeling Technique (VMT) in 1993 [Fang-96].

An important consideration that any new life-cycle should take into account is that there are a number of existing and emerging standards and models that specifically address the software development process, which are growing in popularity. Many organizations are requiring software developers to conform to these software development standards and models which shows the concern for how software is developed. For example, there is the belief that in the near future all software contractors for the U.S. government will be

required to demonstrate a software maturity of Level 3 [Saiedian-95], as defined in the Capability Maturity Model (CMM) [Paulk-93a, Paulk-93b]. Because new standards continue to be developed indicates that people do not fully understand or agree upon the definition of precisely what a good development process is, demonstrating the need for continuing work in defining software life-cycles. An example is the recent ISO/IEC 12207 standard which specifically addresses the software life-cycle [Moore-96, Singh-95].

1.3 Recursive Multi-Threaded Life-Cycle Requirements

There are many goals of software life-cycles, but the primary goal can be summarized as being the definition of a repeatable systematic process for developing quality software within scheduling and budgetary constraints. Like software systems, software life-cycles have requirements that they must satisfy to achieve their goals. Object-oriented life-cycles share many of the same requirements as traditional life-cycles. However, because object-oriented technology has a substantially different approach to developing software, there are many requirements that are more significant or critical to object-oriented life-cycles than traditional life-cycles. These requirements may range from general, being applicable to a large number of projects, to specific, applying to only a small number of projects within a specialized domain. Defining a life-cycle that addresses all of these requirements would be impossible because they may have conflicting goals and/or constraints or add unnecessary overhead to the development process.

RMT addresses the general needs of object-oriented projects but is flexible enough to accommodate the needs of specialized projects. This allows RMT to be compatible with

a large community of developers. To gain a greater understanding of the definition of RMT, the following sections describe the requirements that RMT was designed to satisfy.

1.3.1 Traditional Life-Cycle Requirements

There are a number of requirements for RMT that apply to both object-oriented and non-object-oriented projects. They are:

Monitor progress: RMT should provide the capability to monitor progress and determine completion of the project.

Systematic RMT should provide a systematic process for producing quality software.

Repeatable: RMT should be repeatable for different projects.

Organized: RMT should organize development activities to reduce the complexity of project management, reduce the potential miscommunication between team members, and maintain conceptual integrity of the system during development.

Risk Management: RMT should accommodate the identification and management of risks.

Traceability: RMT should allow developers to trace system requirements to design specifications and to the resulting software.

The primary goal of RMT is to provide developers with a mechanism to monitor the progress of a project during development. Such a mechanism can provide developers with early feedback indicating that there are problems that need to be addressed before they become unmanageable. It can also provide a means for determining when the

development of a project is completed. RMT must provide the capability to monitor progress during development.

Another goal of RMT is to provide a systematic process for producing high-quality software. Software quality may be defined in many terms, depending on many factors. Meyer defines the five most important external qualities of software as correctness, robustness, extendibility, reusability, and compatibility [Meyer-88]. Having a process with a set of well-defined steps or rules to follow for constructing something is much easier than an ad hoc method which bases the success of the project almost entirely on the skill and experience of the developers. Among other things, a systematic process provides the developer with a more accurate experience base for estimating development effort and time, a better metric for gauging progress during development, a better framework for identifying potential problems at an earlier stage, and (hopefully) a higher probability of producing quality software. Another requirement of RMT is that it should be repeatable, so that it can be applied to many projects rather than discovering a new process for each new project. This saves the developer valuable time and effort.

As Brooks describes, software is by nature inherently complex [Brooks-95]. More specifically, it is the construction of the conceptual representation of the software that introduces the complexity, not the actual realization of the concept. Part of this complexity can be attributed to the management of the activities during the development process. How development activities are organized can have a drastic impact on the effectiveness of the development of software. Projects of significant size tend to involve larger teams. Larger teams increase the potential for communication problems and decreases the

conceptual integrity of the system simply because there are more people involved in the process. To help address these complexities, RMT should provide a framework to organize development activities in such a way that the potential for these problems is reduced.

RMT must specify an activity (or activities) to identify and manage potential problems, or risks, that might impact the development process. This is commonly called risk management and is an important activity of the development process. It is better to identify potential risks and plan for them before they happen rather than ignoring them and reacting to them after they occur. Risk management is more than simply identifying potential risks, but also includes monitoring the of risks during development, mitigating or avoiding risks (if possible), and carrying out some contingency plan should risks occur.

Once a software system has been implemented, it is essential to verify that the resulting system meets the requirements of the user. Therefore, RMT must facilitate the verification of system requirements to the produced software. While the methodologies used during development (i.e., requirements analysis, analysis, design, etc.) and resulting documentation usually facilitates this, RMT should also provide well-known paths of communication between team members to make this process easier.

1.3.2 Object-Oriented Life-Cycle Requirements

As previously mentioned, there are a number of characteristics that are more critical to the development of object-oriented software than traditional, non-object-oriented, software. While non-object-oriented projects may also strive for these qualities as well, they are essential to object-oriented software. These RMT requirements are:

- Iterative development:* RMT should support an iterative development process.
- Parallel development:* RMT should support the overlap of development activities.
- Reuse:* RMT should support the reuse of design information (design patterns) and source code.
- Maintenance:* RMT should accommodate maintenance as part of the software life-cycle.

Many methodologists agree that successful development of object-oriented software involves iteration. Gilb believes that software evolves over a period of time, similar to the development of complex systems, such as biological organisms. [Gilb-88] This is called evolutionary development, of which iteration is a key concept. Booch has observed that two traits, well-managed iterative and incremental development life-cycles and the concept of a strong architectural vision, were present in virtually all successful object-oriented systems he had encountered, and absent from unsuccessful systems [Booch-91].

Iterative life-cycles allow the incremental development (and delivery) of a system by producing many versions of the system, each more (functionally) complete than previous versions. While there are a number of benefits of iteration, the most significant is its adaptability to change. Because there are frequent incremental versions of the system, iterative life-cycles allow potential problems or changes to be identified earlier in the development cycle where the amount of effort to correct the problem is smaller, rather than late in the cycle. For example, consider a project where at the beginning of the

project the perceived objective is Objective A. At some point during development either the users or developers realize that the actual objective is not really Objective A, but Objective B. A traditional process with a single delivery of the system will not discover that Objective A is the incorrect objective until the software is completed, requiring a significant amount of effort to be expended to adapt the software to satisfy Objective B. An iterative process, however, could help identify the changed objective and react to the change at an earlier point in development, reducing the amount of effort required to reach Objective B. Figure 1.1 illustrates this example.

Because iteration is an essential requirement for developing successful object-oriented software, and because of the additional benefits, RMT must be an iterative-based process.

Another characteristic of object-oriented development, that is less pronounced in structured approaches, is that there tends to be overlap between activities during development. The concept of a class provides a common conceptual unit, or vocabulary, that is used throughout development activities (e.g., analysis, design, and coding), and each activity in an object-oriented life-cycle produces a more complete definition of a class. As a result, the division between the completion of one activity and the beginning of another becomes less distinct. For example, Berard points out that the “gap” between object-oriented requirements analysis and object-oriented design is very narrow when compared to the “gap” between structured analysis and structured design [Berard-93]. Requiring each development activity to be completed before beginning another activity

would be an unnecessary restriction to the development process of object-oriented software. Therefore, RMT must support parallel development.

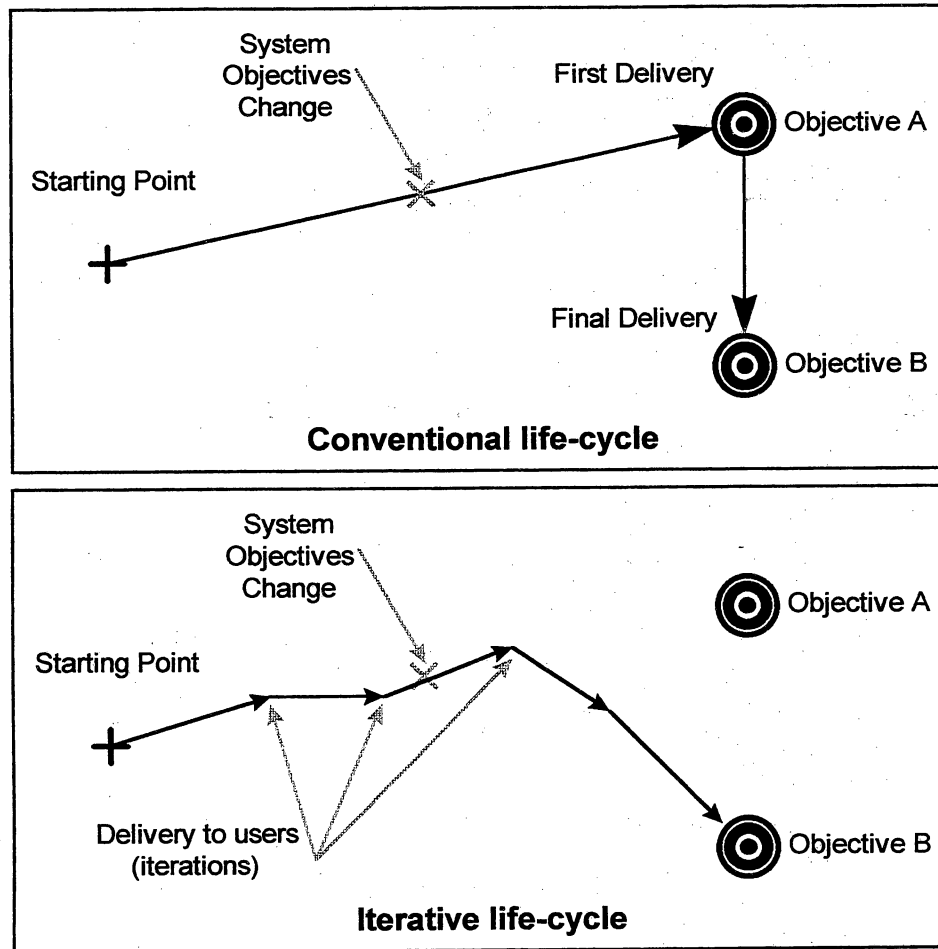


Figure 1.1: Conventional vs. Iterative Life-Cycles

Software reuse has been a goal of software engineering long before object-oriented technology became popular. One attraction for using object-oriented technology is its potential for producing reusable software components. While object-oriented programming languages may make the procedure of building reusable software components easier than procedural programming languages, it is still more costly to build

reusable components. Yourdon estimates that reusable components take twice the effort as “one-shot” components [Yourdon-92].

Another level of reuse that has only recently emerged in the shadow of object-oriented technology is design patterns. Design patterns are an abstraction of source code that contains (proven) design information for a solution to a particular problem. When compared to source code reuse, design patterns are less effective because they still need to be realized into some form of code and tested. However, given all of the difficulties associated with source code reuse, design patterns may be more useful because they have a greater potential of actually being reused. Because of the potential benefits of reuse, RMT should accommodate the evaluation and integration of both source code and design pattern reuse in the development process.

Many software life-cycles consider the initial development and deployment of a software system and maintenance as separate activities. Software maintenance may involve more than simply corrective maintenance, or “bug fixing”, it may also include adaptive maintenance, perfective maintenance or enhancements, and preventive maintenance or reengineering [Swanson-76]. Maintenance can account for more overall effort during the life-cycle of a software system than any other activity, an average of 67%, in fact [Lientz-78, Zelkowitz-79]. For these reasons, RMT should make accommodations for maintenance as part of the software life-cycle.

1.4 Capability Maturity Model (CMM)

There has been much effort in the software engineering community to define standard practices and methods for software development to improve how software is

developed. One of the most recognized efforts is the Capability Maturity Model (CMM). The CMM has the goal of improving software quality by defining various levels of development process maturity. While the CMM does not define or advocate the use of a particular software life-cycle, it does define some characteristics that must be present in a software life-cycle in order to comply with their requirements. Because the CMM is growing in acceptance among the software community, RMT should conform to CMM requirements as much as possible.

1.5 Structure of Thesis

This thesis is organized into five chapters and one appendix. This chapter presents an introduction and motivation for the work proposed in this thesis, a summary of development life-cycle requirements, and a brief description of the CMM. The second chapter summarizes a number of existing software life-cycles and compares them to the requirements outlined in the first chapter. The third chapter presents the proposed RMT software life-cycle, providing a concise definition of the individual components of the life-cycle. Chapter four presents an example of how RMT can be applied to a specific project. The fifth chapter provides some conclusions about RMT (its strengths and weaknesses), how RMT applies to the CMM, and future directions that should be explored for RMT. Appendix A contains a glossary of terms used throughout this thesis.

Chapter Two-Existing Software Life-Cycles

This chapter presents a brief description of a number of existing software life-cycles, their limitations, and/or any conflicts these life-cycles have with the requirements outlined in chapter one. The life-cycle descriptions are not intended to be complete by any means. There are a large number of existing software life-cycles, but this chapter only presents those life-cycles that were deemed relevant to RMT. They are included either for historical purposes or their relevance for comparing/contrasting them with RMT.

2.1 Taxonomy of Software Life-Cycles

The life-cycles discussed in this chapter are divided into three categories: non-object-oriented life-cycles, object-oriented life-cycles, and “second-generation” object-oriented life-cycles. The non-object-oriented life-cycles are included for historical purposes to help identify why many traditional life-cycles are inappropriate for object-oriented projects. The category of “second-generation” life-cycles refers to life-cycles that integrate and/or extended existing approaches. Table 2.1 outlines the life-cycles discussed.

| Classification | Life-Cycle |
|---|---|
| Non-Object-Oriented Life-Cycles | Waterfall Model |
| | Spiral Model |
| Object-Oriented Life-Cycles | Round-Trip Gestalt Design |
| | Recursive/Parallel Model |
| | Fountain Model |
| | Chaos Model/Life-Cycle |
| “Second-Generation” Object-Oriented Life-Cycles | McGregor and Sykes |
| | Visual Modeling Technique (VMT) |
| | Methodology for Object-oriented Software Engineering of Systems (MOSES) |

Table 2.1: Taxonomy of Software Life-Cycles

2.2 Waterfall Model

The waterfall model [Royce-70] is probably the most widely recognized software life-cycle. It is a linear life-cycle model with a number of development activities that are performed sequentially. Before an activity can begin, the previous activity must be completed.

The waterfall life-cycle is a dramatic improvement over the ad hoc build-and-fix method that was commonly employed before its introduction. Unfortunately, there are many problems and limitations with the waterfall model. The most significant problem is that software development is rarely a sequential process. This does not accommodate changes during development, requires all of the system requirements to be completely and accurately specified at the beginning of the project, and results in inefficient use of

personnel resources. Pressman [Pressman-97], Brooks [Brooks-95], McGregor and Sykes [McGregor-92], and others all confirm limitations of the waterfall life-cycle.

2.3 Spiral Model

The spiral model [Boehm-88] is a risk-driven software life-cycle that iterates through four basic activities: objective assessment, risk assessment, product development, and planning. Development starts at some central point, from which development proceeds outward from the center (i.e., like a spiral), passing through each of the four activities or quadrants. As the spiral gets larger, so does the cumulative cost. Each cycle in the spiral model builds the next-level product of the resulting system. These products correspond to the commonly identified life-cycle activities (e.g., requirements, design, etc.).

Even though the spiral model appears to be an iterative life-cycle, it is not truly iterative because there is a finite number of circuits and each circuit really corresponds to a development phase or activity. For example, implementation occurs during a single circuit. What makes the spiral model appear to be iterative is the fact that within each circuit similar activities, such as planning, determining objectives, evaluating risks, etc., are repeated in each circuit. This is ill-suited for the iterative requirements of an object-oriented life-cycle. In addition, the spiral model does not support the overlap of activities during development. The spiral model is also applicable only to large-scale projects [Boehm-88], making it unfeasible for small to medium scale projects.

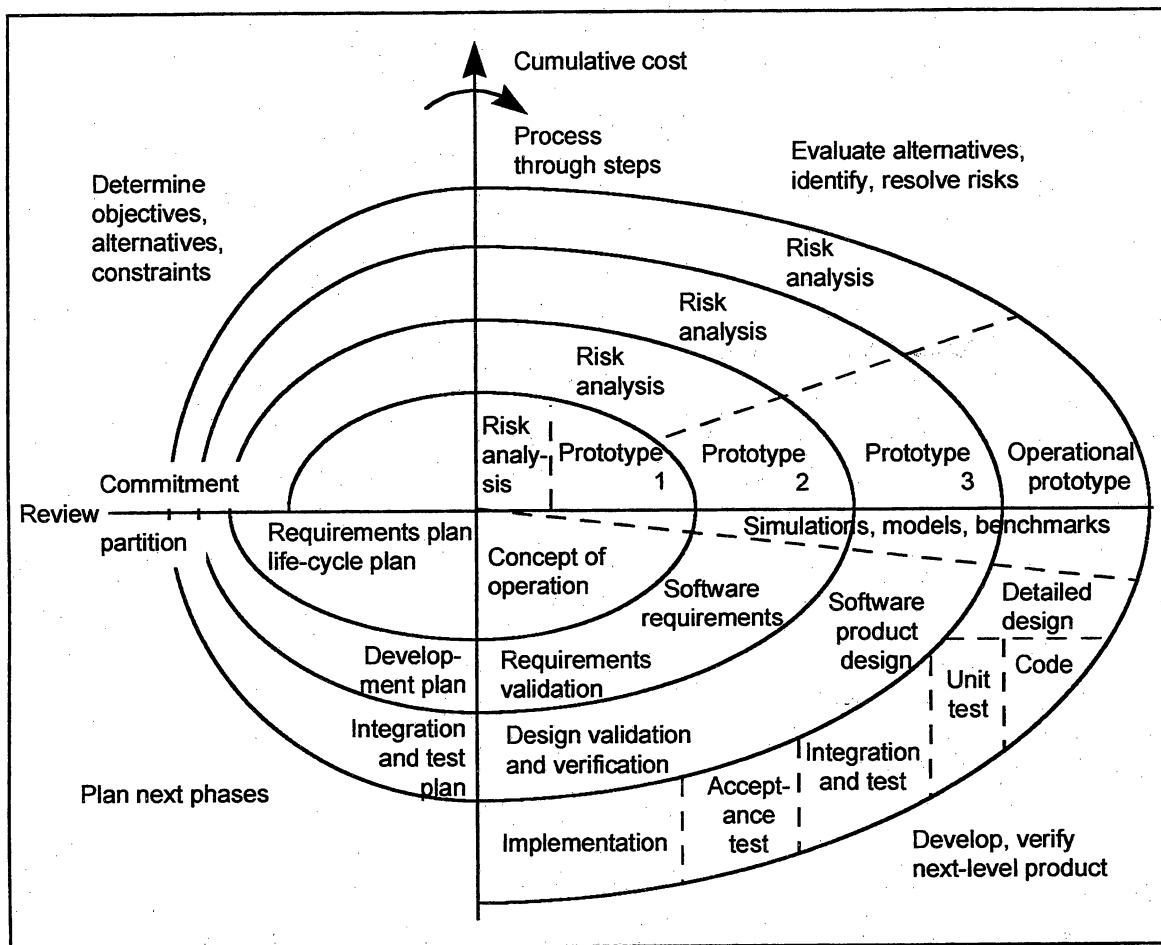


Figure 2.1: Spiral Model

2.4 Round-trip Gestalt Design

The round-trip gestalt design [Booch-91] is a design method based upon the fact that the more that is known about a problem, the easier it is to solve. When a designer is confronted with a new problem where they have limited or no experience, the best they can do is to make an initial attempt at the design, step back and analyze the design, then make improvements based upon new understanding of the problem. This process is repeated until the designer is satisfied with the completeness and correctness of the design. This is the round-trip gestalt design.

Although the round-trip gestalt design is a design method and not a life-cycle, it's essence has been used for comparison to iterative software life-cycles. In fact, Booch suggests that it is the foundation of the process of object-oriented design [Booch-91].

2.5 Recursive/Parallel Model

The recursive/parallel life-cycle can be caricatured as “analyze a little, design a little, implement a little, and test a little.” [Berard-93] Rather than being a life-cycle that was first defined then applied to projects, this software life-cycle evolved from software engineers applying object-oriented technique to real projects. Berard points out that any significant software engineering effort will involve both iteration and overlap as well as addressing requirements of different levels of abstraction at different times during development. This life-cycle more accurately reflects theses realities of software engineering and simply formalizes the concepts and techniques already used by engineers.

While the recursive/parallel life-cycle is a “top-down” approach, which Berard states is very often a noticeable flavor to the overall approach for projects, it does support compositional, or bottom-up, techniques. The systematic steps in the recursive/parallel life-cycle are:

- (1) “Systematically decompose a problem into highly-independent components,
- (2) re-apply the decomposition process to each of these components to decompose them further (if necessary)—this is the ‘recursive’ part,
- (3) accomplish this re-application of the process simultaneously on each of the components—this is the ‘parallel’ part, and

(4) continue this process until some completion criteria are met.” [Berard-93].

The analysis step requires that the system requirements be understood, propose a “high level” solution for the requirements, and demonstrate that the proposed solution meets the user’s needs. The design step involves the definition of the component interfaces, making decisions about how each component will be implemented, the identification of any necessary additional components, and describing any necessary programming language relationships. The implementation step requires the implementation of the component interfaces, the implementation of the algorithms describing the component interactions, and the implementation of the internals of components which can not be further decomposed.

While this life-cycle addresses many of the fundamental requirements of object-oriented life-cycles (outlined in chapter one), it lacks the detail necessary for the direct application to a project, leaving too much of the process organization up to the developer to define. For example, it does not address the management or organization of the “recursive” or “parallel” elements of the life-cycle, risk management, or planning activities.

2.6 Fountain Model

The fountain model is an object-oriented software life-cycle that supports a high degree of overlap and iteration during development [Henderson-Sellers-90]. The general flow through development activities proceeds from analysis through design to implementation, with iterative cycles across several or all of these phases. Development during any phase may iterate back to any previous phase. The system life-cycle may be composed of a number of separate class, or clusters of classes [Meyer-89], life-cycles. The

number of development phases included in each model varies upon the application of the life-cycle. For example, the fountain model for module development may consist specification, design, coding, and testing phases while system development contain additional design, requirements analysis, and testing phases. Because the system view of the life-cycle may be composed of many other class life-cycles, class clusters may be developed independently of and in parallel with other class clusters.

Like the recursive/parallel life-cycle, the fountain model accommodates the iterative and incremental requirements of object-oriented projects, but it is lacking in detailed descriptions of how the overall development activities and team members are organized. It is almost too flexible. The danger of such flexibility is that the development process can become undisciplined where developers proceed almost randomly between phases. This makes project management and progress monitoring very difficult, if not impossible.

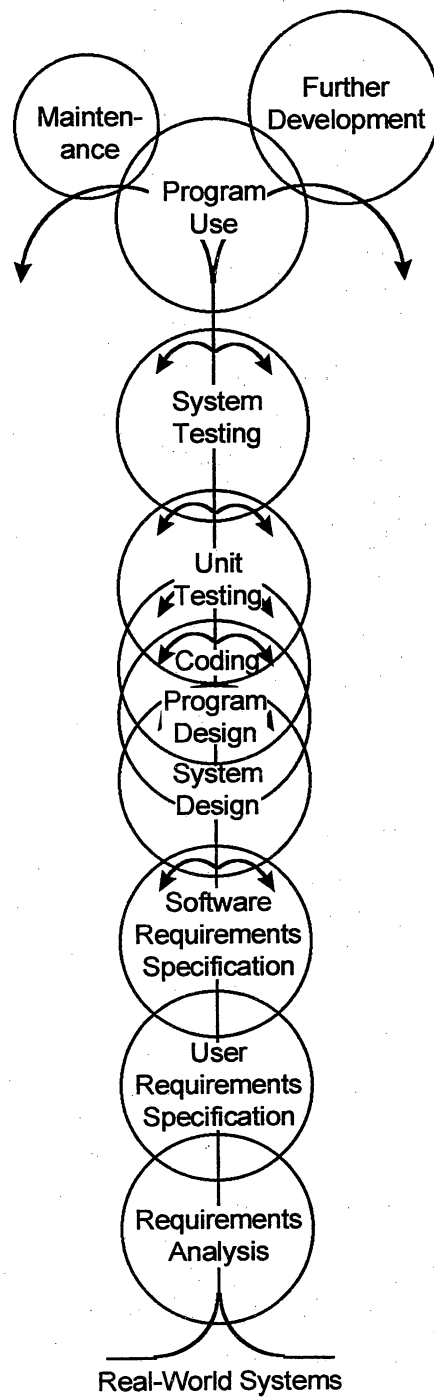


Figure 2.2: Fountain Model

2.7 Chaos Model/Life-Cycle

Raccoon [Raccoon-95] believes that because of the complex nature of developing software, simple models can not be imposed upon it. To represent the realistic nature of software development, the chaos model describes software development as a linear problem-solving loop combined with fractals. The linear problem-solving loop consists of four stages: problem definition, technical development, solution integration, and status quo (i.e., the current state of the system). In theory, the fractal problem-solving loop is simply the linear problem-solving loop where each phase contains an identical problem-solving loop. In reality, however, there are a number of influences during development that make the localization of recursive problem-solving loops to higher-level problem-solving loop phases difficult. Each phase in the chaos life-cycle is expressed as a fractal. Because of the recursive nature of fractals, Raccoon points out that each phase occurs in all other phases and that each phase is a complete life-cycle itself. The life-cycles phases then blend together resulting in an “amorphous flow of emphasis” [Raccoon-95] rather than separate, distinct phases.

From the perspective of understanding the nature of software development, the chaos model and chaos life-cycle provides developers with a better understanding of the complexities of software development and the factors influencing development. For application to real world projects, however, the chaos life-cycle is impractical because it does not provide enough organization of development activities. This makes progress monitoring, planning, communication, etc. difficult for developers because there is a very complex and unorganized structure to the life-cycle phases or activities.

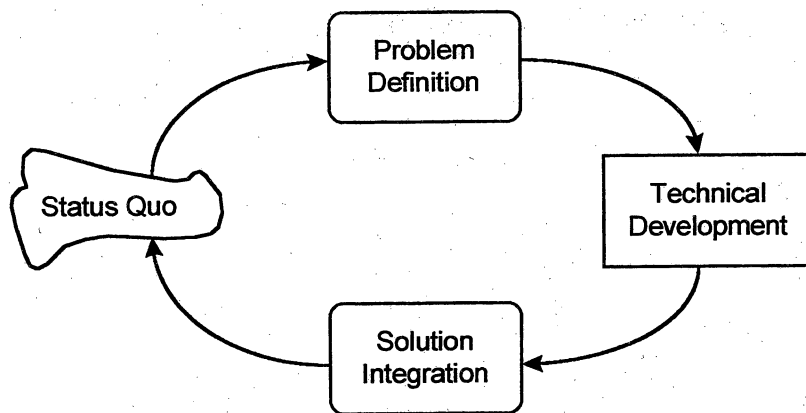


Figure 2.3: Linear Problem-Solving Loop

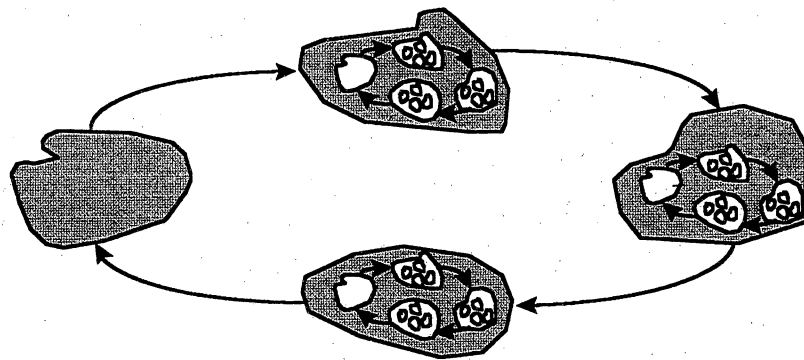


Figure 2.4: Fractal Problem-Solving Loop

2.8 McGregor and Sykes

McGregor and Sykes [McGregor-92] have proposed a software life-cycle that emphasizes reuse and the support for the object-oriented paradigm. They divide the development life-cycle into two independent, and orthogonal, life-cycles; the application life-cycle and class life-cycle. The reason for dividing the two is to produce more reusable classes. They believe that this division allows for a complete description of the classes to be built without regard for the system being developed, making the classes more reusable. The class life-cycle is very similar to the fountain model, but accounts for the reuse of

existing classes, evolution from an existing class, and the development of a class from scratch. The detailed representation of the application life-cycle consists of a series linear steps, although the actual development process is not (figure 2.5). The visualization of the overall process is described by the “fractal model.” (figure 2.6) which is based upon Brian Foote’s “fractal model” proposed at an OOPSLA ’91 research workshop on reuse.

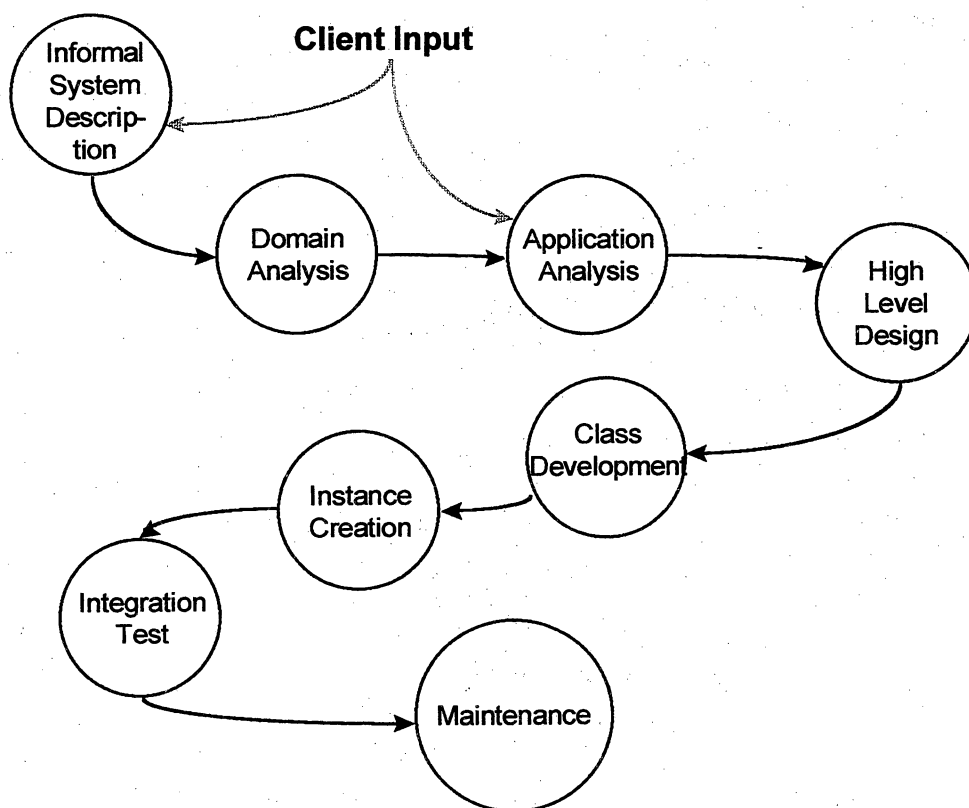


Figure 2.5: McGregor and Sykes Application Life-Cycle.

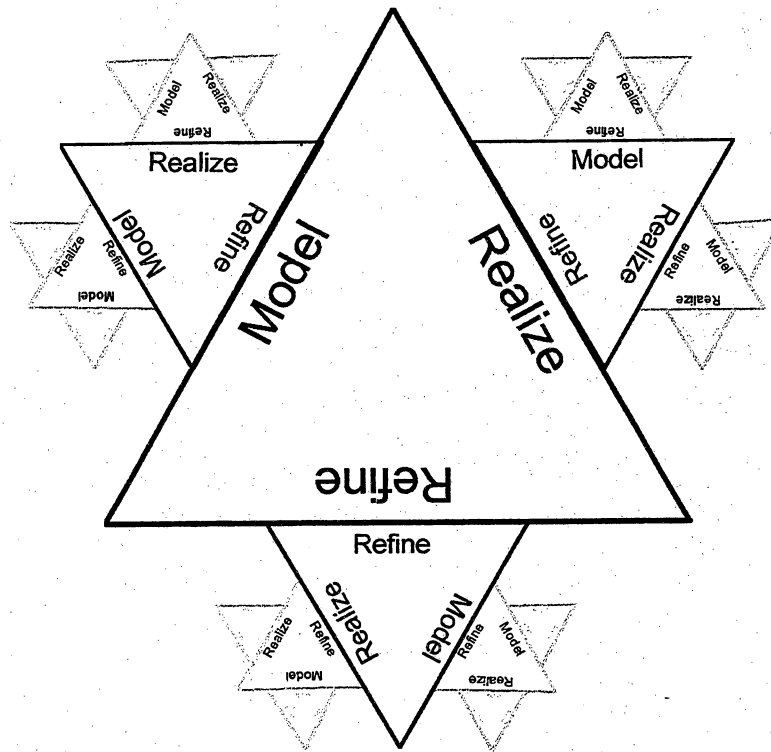


Figure 2.6: McGregor and Sykes Fractal Software Development Process.

While the application and class life-cycle descriptions suggest that they are iterative in nature, the iterative or incremental steps in the process are only detailed in the class life-cycle and not in the application life-cycle. Even though the activities in the application life-cycle phases are discussed, the overall application of the life-cycle is not presented, leaving the life-cycle definition vague and incomplete. The central focus of the McGregor and Sykes process is to build reusable objects, which requires that two versions of a class to be implemented when developing classes from scratch; an abstract class and a concrete class. The intent is that the abstract class embodies the essence of the class, independent from an application specific details, and the concrete class is derived from the abstract class and addresses the application requirements. While this may result in more

reusable objects, it requires significant effort to implement. There may be many projects where developing truly reusable classes is not a priority, making this process too expensive.

2.9 The Visual Modeling Technique (VMT)

The Visual Modeling Technique (VMT) is a complete object-oriented development life-cycle that is based upon existing and proven methodologies and techniques. The core techniques used are the Object Modeling Technique (OMT) [Rumbaugh-91], Jacobson's use cases [Jacobson-92], Wirfs-Brock's Responsibility Driven Design (RDD) [Wirfs-Brock-90], CRC cards [Wirfs-Brock-90, Wilkinson-95], event trace diagrams, object types, and pre- and postconditions. The product life-cycle consists of a business planning, development, and packaging/delivery phases. The development phase of a product life-cycle is divided into a number of increments, each which may further be divided into a number of iterations. Each increment consists of a planning period followed by a production and assessment period. The production period consists of the common software life-cycle phases analysis, design, coding, and testing.

VMT supports the iterative and incremental nature of object-oriented software projects. It also supports reuse and project management activities. The main emphasis of VMT, however, is in how the previously mentioned methodologies are applied during each of the production periods during the life-cycle. While this may be very useful and productive for individuals fluent with these methodologies, others may not be familiar with them or may be unwilling to change to these methodologies, making VMT an inappropriate life-cycle.

2.10 Methodology for Object-oriented Software Engineering of Systems (MOSES)

The Methodology for Object-oriented Software Engineering of Systems (MOSES) is a complete object-oriented software life-cycle that has evolved from previous work presented by both Henderson-Sellers and Edwards [Henderson-Sellers-94]. In addition to the delineation of the process phases, it also supports a set of graphical and textual notations. The MOSES life-cycle recognizes two separate life-cycles: the product life-cycle and the process life-cycle. The product life-cycle is divided into two distinct periods of a software system's lifetime, the growth period, where the initial system is constructed, and maturity period, where the system is maintained and enhanced. Both the growth and maturity periods consist of three phases. These are the business planning stage, the build stage, and the delivery stage. The build stage is where the software is actually constructed and involves the application of the process life-cycle.

The process life-cycle is an iterative development process (IDP) that is based upon the fountain model [Henderson-Sellers-90]. It recognizes five phases of development: planning, investigation, specification, implementation, and review. Each phase has well defined goals, performed tasks, and deliverables.

While MOSES hints at the problem of decomposing system development into smaller problems, it only discusses one level of decomposition by decomposing the entire system into a number of subsystems which may be developed in parallel. MOSES does not advocate the recursive application of the life-cycle upon each decomposed subsystem. MOSES also uses a custom notation for diagramming designs which integrates a number

of other notations. The use of a custom notation may be unacceptable for some developers.

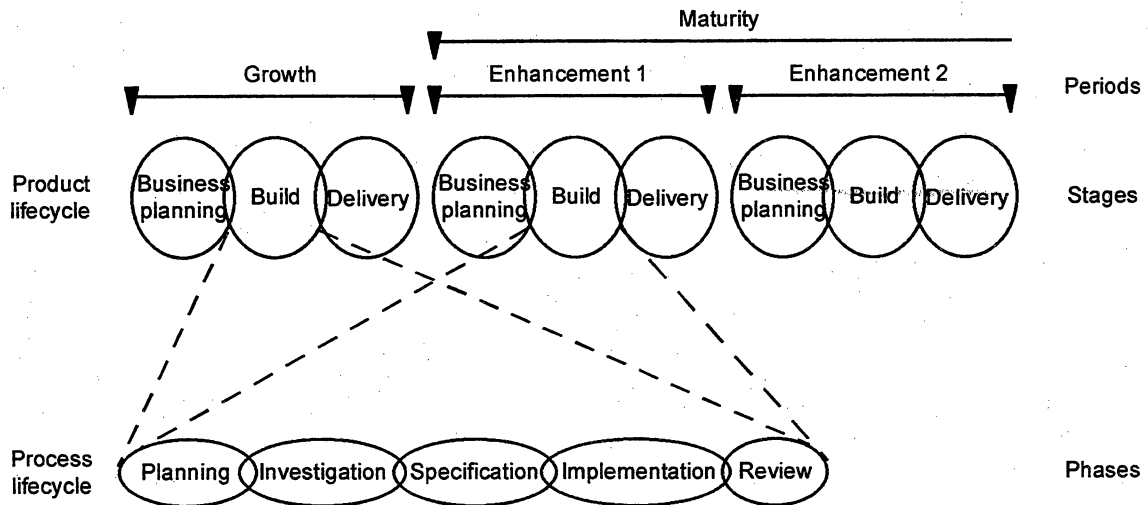


Figure 2.7: Overall MOSES Life-Cycle

2.11 Common Limitation

In addition to any individual limitations or deficiencies noted, each of the object-oriented life-cycles outlined in this chapter contain a common limitation; they do not explicitly account for monitoring progress during the development process (non-object-oriented life-cycles are not considered because they do not address the requirements of object-oriented development, and the round-trip gestalt design is excluded because it is not a life-cycle). Monitoring progress is an important part of managing a project because it helps the project manager determine whether or not the project will meet its schedule (and scheduling constraints are a requirement common to most all projects). Estimating progress can be difficult. Without some technique for estimating progress, estimates are simply best guesses based upon the opinions of the developers. Personal opinions will vary

between individuals and the accuracy of the estimate depends upon their education, experience, skill, and luck.

Even though estimating progress is not explicitly supported by the mentioned life-cycles, additional methods could be used. However, the organization of the development process in each of these life-cycles makes estimating progress fundamentally difficult (but not necessarily impossible) for one of two reasons. The first reason is that some life-cycles are too flexible by allowing development to proceed almost randomly between activities making it difficult to determine the current state and progress of development. The fountain model, chaos life-cycle, and McGregor and Sykes are examples of this flexibility.

The other difficulty imposed by some life-cycles, such as VMT and MOSES, on estimating progress is that the smallest unit of management is an iteration, which makes estimating progress difficult (and potentially inaccurate). An iteration in these life-cycles represents a version of the entire system. The progress for the overall project is based upon the individual estimates of the many components comprising the overall system. Each component represents a certain percentage of the overall effort to implement the system, so the estimate for each component must be weighted relative to its overall significance to the system. Because the iteration is the smallest unit of abstraction, estimates for all the software components have to be evaluated, weighted, and compiled at one abstraction level to produce an overall progress estimate. Analyzing progress estimates for all of the software components together forces a developer to analyze too many logical entities simultaneously to evaluate/interpret them effectively.

Chapter Three-The Recursive Multi-Threaded (RMT) Life-Cycle

The recursive multi-threaded (RMT) life-cycle is designed to accommodate the needs of developing systems using object-oriented techniques and to facilitate the monitoring of progress during development. The previous chapters discussed the motivation and requirements of RMT, and summarized a number of existing software life-cycles and some of their limitations. This chapter presents the fundamental concepts and definition of the RMT software life-cycle.

Many of the underlying concepts and techniques of RMT are also found in existing life-cycles (e.g., the spiral model [Boehm-88] and the recursive/parallel model [Berard-93]), but the presentation and implementation of those concepts differentiate RMT from these life-cycles. Even though techniques used by RMT, such as iteration and recursion, have also been proposed in existing life-cycles, what differentiates RMT from existing life-cycles is the use of a development “thread” as a conceptual unit to organize development activities and to monitor progress. RMT is a milestone-based, iterative life-cycle that supports incremental and parallel development. It uses a divide-and-conquer technique to system implementation, supports multiple levels of information abstraction, and encourages the use of open-ended architectures. The use of threads to organize development helps provide a form of control to the complex nature of object-oriented software development (often interpreted as chaotic).

3.1 Process Concepts

In a nutshell, RMT consists of a number of threads which implement some software system. Each thread is an abstraction which represents the implementation of

some portion of the overall software system. A thread consists of a set of activities that are performed in some order to implement a software component (which may be a class, module, or subsystem), and may be iterated many times. A thread may also spawn child threads which implement some portion of the software component of its parent thread. Since a thread may be composed of other threads, there may be many threads executing simultaneously at any point of the development process. Because of this hierarchy of threads, RMT is a divide-and-conquer process and as described in later sections, the hierarchy of threads divides the system implementation into multiple levels of abstraction. Supporting multiple levels of abstraction provides a framework for monitoring progress during development.

There are a number of essential concepts that define the RMT process. Specifically, they are threads, iteration, recursion, and reuse. The following sections describe each of these concepts in detail.

3.1.1 Threads

The central concept of RMT is a thread. Most everything within RMT is defined in terms of a thread. Threads are most commonly discussed in the context of programming languages and operating systems. In this context, a thread is commonly a single path of execution within a program, where multiple threads may be executing the same program simultaneously. This allows for parallel execution within a program. This is different from processes within an operating system because each of the threads shares the same program instructions and memory. A more detailed discussion on programming threads exists in [Lewis-96].

An RMT thread consists of a set of activities, or phases, that have well-defined goals, inputs, and outputs. These activities are not unique to RMT but are present in many other software life-cycles. An RMT thread is composed of planning, requirements analysis, analysis, design, implementation, testing, and quality assurance phases. These activities are generally performed in a sequential order, although there may be overlap between some phases. Unlike traditional, sequential life-cycle models, certain thread phases may begin prior to the completion of the preceding phase. The most common overlap of phases occurs in the analysis, design, implementation, testing, and quality assurance phases.

While the analysis, design, implementation, and testing phases may overlap with each other, Berard [Berard-93] points out that software quality assurance (SQA) is an activity that occurs during the entire life-cycle and not just at the end. SQA does not only consist of testing, it may also include requirement verification, insuring consistency between analysis, design, and implementation, performing design and code inspections, etc. The Software Engineering Institute [Paulk-93a] recommends a set of SQA activities that should be carried out by a group independent from the developers. Because these activities and the individuals performing these activities are independent from (yet closely tied to) the development activities, SQA could be considered its own process with a separate life-cycle that occurs in parallel with the development life-cycle. A sample SQA life-cycle might consist of walkthroughs and risk analysis during the requirements analysis phase, inspections and risk monitoring/management during design and implementation phases, and testing after implementation is complete. This is not the only or best SQA life-cycle. Because the SQA activities used by organizations may vary greatly, RMT defines a

minimum set of SQA activities but allows for additional activities during all development activities.

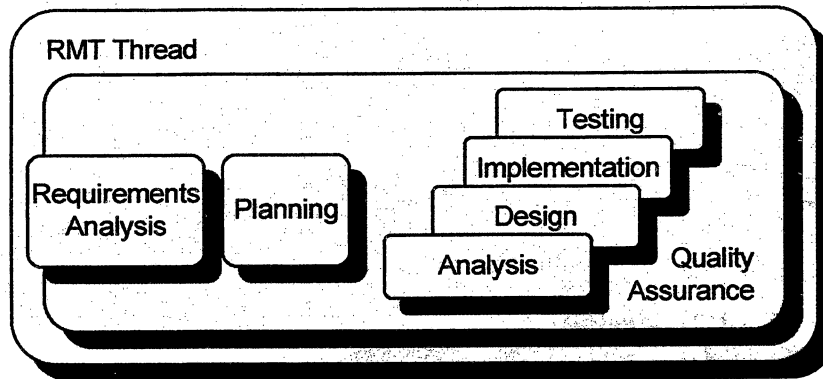


Figure 3.1: RMT Thread Activities/Phases

Each thread has a team of individuals (one or more) who perform activities to implement software components to satisfy the requirements for that thread. Within a thread team there is one individual, the thread manager, who is responsible for the software component(s) built by the thread. Developers may work on many different threads and thread managers may manager more than one thread.

The same step-by-step process defined by a thread is applied to many different parts of a project by many different developers with different skills and responsibilities. For example, the same thread abstraction used by an engineer to implement a single class is also used by the project architect for the conceptual view of the entire system. This is analogous to threads in programming languages where multiple threads share the same set of instructions. In RMT, these shared instructions are simply the steps, or activities, that are performed during a thread.

3.1.1.1 Iterative/Evolutionary Development

As Booch, Gilb, and others have described, there is a need to support iteration and incremental development within an object-oriented development life-cycle. There are a number of reasons why iteration may occur during software development (and why a software life-cycle should accommodate it). One reason is that it is simply easier to partition development into smaller, more manageable pieces. A common method for incrementally developing a class is to implement the complete interface with methods that do nothing (a stub), then incrementally implement (or extend) each of the stubs.

When given a set of requirements for a software component (whether they are for an entire system or for a single class), the development of the component should be partitioned into a number of incremental releases, distributing the requirements among the incremental releases. The requirements should be prioritized according to an effectiveness/cost ratio and scheduled so that the highest ranked requirements are included in the earliest releases [Gilb-88]. It is possible that the planned iterations may change during the course of development. Planned iterations may be removed because system requirements may be deleted or new iterations may be added due to new requirements or the modification of existing requirements. In addition, if technical problems occur, such as design or implementation flaws, new iterations may be required to resolve the flaws.

These incremental releases do not need to be given to the end user or other team members, but may simply be used as an internal development milestone. In fact, an incremental release may not even satisfy any of the given requirements. Early project increments may simply implement a basic system architecture or framework that the

remainder of the software system will be built on. Thread iterations may also be used as a way to explore and further define vague or incomplete requirements, evaluate potential risks, or to prove/disprove crucial design decisions. When given vague requirements or the design for a critical component, a thread iteration may simply implement a prototype to clarify requirements or as a proof-of-concept for a design specification. This prototype can be included as a thread iteration during the planning phase for the thread.

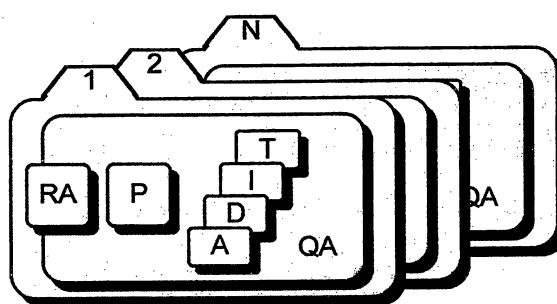


Figure 3.2: RMT Thread with N-Iterations

In addition to simply partitioning a problem into smaller pieces, iterative life-cycles are well-suited for handling changes during development, help identify differences between the defined system requirements and the “true” user requirements early in the development process, provide a realistic metric for measuring progress, and help prevent defects from becoming overwhelming.

Changes during the development process require that developers “backtrack” to some previous point in the development process, modify or correct some problem, then continue development along the same, or different, path. Many life-cycles do not adequately handle changes in requirements, design, etc. during the development process, viewing them as a negative influences that should be avoided. Brooks [Brooks-95] feels

that change is simply a fact that we should accept and accommodate rather than try to ignore or attribute to poor decision making. Technical problems, such as poor design or implementation decisions, and non-technical problems, such as misuse of technology and personnel conflicts [Raccoon-95], may require development phases be revisited to correct the errors. In sequential methods, revisiting previous development activities to correct errors or accommodate changes tends to incur significant costs because it is not part of the planned sequence of events. By expecting and planning for development phases to be repeated, iterative life-cycles are more accommodating to change.

Because many time the users (and developers) do not completely understand the system requirements at the beginning of the project, the system requirements may change during development. Incremental releases of the software can be given to the users to solicit feedback. Users are able to identify incorrect or missing requirements early in the development process rather than after the final software is delivered. This prevents developers from expending significant effort building the software to incorrect specifications which will require additional effort to modify the software to the new specifications later.

At each incremental release during an iterative life-cycle, the actual development progress can be compared with the project schedule and the schedule can be adjusted accordingly. Because this is done frequently, it provides the developers and managers with a more accurate view of the development and estimated completion based upon the realities of what has been currently implemented.

Iteration also help keep software defects at a more manageable number because as each iteration is implemented, defects are resolved before the iteration is complete. This prevents a tremendous amount of defects from having to be resolved at once, like in a single-release approach. Because defects are resolved at the end of each iteration, each iteration produces a working/tested component making the software more stable earlier in the development process.

Because development of the software components is divided into multiple iterations, most iterations are based upon some existing version of a software component. Since the majority of thread iterations are based upon existing software, the only distinction between the initial thread iteration and subsequent iteration is that there is no existing analysis/design information, source code, etc. to be taken into consideration during the initial iteration. In fact, there is no reason why the initial thread iteration can not be based upon an existing software component, it just requires the developer to review the existing software component just like during subsequent thread iterations. Because of this, developers can improve or extend existing software components at any time during a life-cycle, whether the component is currently under development or has already been delivered to the user and is “development-frozen”. In RMT, maintenance of a software component (or system) is no different than the initial implementation of that component, it simply requires new iterations to implement additional requirements for the existing component. Even in the normal course of the development of a software component, new requirements may be added after development has begun and before it is completed. Maintenance is no different. Viewing maintenance as part of the system life-cycle by

continually repeating development phases (i.e., iteration a thread) makes RMT cyclic in nature (thus, the term life-“cycle”) because it has no end-point.

Care should be taken when planning the number of iterations a thread should have. Using too many iterations to implement a component can have detrimental effects on productivity, requiring more effort to manage the iterations themselves than is saved as a result of using iteration. Iterations should only be planned when the benefits of dividing the development of some component into a number of iterations is greater than the cost of managing the iterations themselves. The criteria used for determining how many iterations to use for a particular thread depends greatly on the nature, complexity, and functionality required of the component to be implemented.

To help guard against developers making poor judgments and scheduling excessive iterations, a guideline for determining how many iterations to use is that each iteration should represent a significant portion of either the overall effort to implement the component or a significant portion of the overall functionality of the component. For example, each iteration should represent no less than 15-20% of the overall effort or functionality of the component (i.e., a maximum of 5 to 7 iterations). Exceptions may be made to this guideline for very complex components. Microsoft, for example, uses three or four project milestones (similar to an iteration) for developing products. [Cusumano-95] Another safeguard is to have a peer review by a group of developers of the estimated number of iterations for a thread during the thread planning phase.

3.1.1.2 Recursion

RMT threads, like threads in programming languages, may create child threads. Within an RMT thread, the implementation phase may simply be the realization (coding) of a simple software component (a class) or a complex software component (a subsystem). For non-simple software components, the implementation phase may actually be the recursive application of a number of more specialized threads, where each child thread implements a particular portion of the complex software component.

Each RMT thread begins with a given set of requirements for a software component that the thread must implement. These requirements may be in varying levels of abstraction, ranging from very high-level (for an entire system) to very specific (for a single class). As previously mentioned, these requirements may be prioritized and implemented in various thread iterations. Within a single thread iteration, the implementation phase begins when enough design information has been defined from analysis and design phases to specify what needs to be implemented (the preconditions of the implementation phase are specified later). If the design information is the specification for a small-grained component (a class or group of classes) then the implementation phase results in the actual coding of the component. If, however, the design is for a higher-level component, then the current design must be further detailed to identify and define all of the classes required to implement the higher-level component(s). To make this process of specialization more manageable, the design of each higher-level component is decomposed into smaller cohesive groups and new, more specialized, threads are spawned to satisfy each of these groups of requirements (i.e., divide-and-conquer). Each of these child

threads follow the same rules as its parent thread; they may iterate many times and they may have a number of child threads themselves. The implementation phase of a given thread is completed when all iterations of all its child threads have been completed or it has been terminated prematurely because of some failure.

Because threads may create other threads, there may be any number of threads that are being “executed” at any given time, each of which may be in a different phase. In this sense, an RMT thread is similar to a high-level programming language thread. In addition, all development initiates from a single thread, the root, which represents the entire system. All other threads are spawned, either directly or indirectly, from the root thread.

While recursion has its benefits, it also has its pitfalls. Anyone who has written recursive programs has undoubtedly discovered this at one time or another when they incorrectly code the exit condition and their program fails to terminate. While recursion can be an eloquent solution to a problem, it adds additional overhead. In programs, recursion requires additional resources (memory). In RMT, recursion requires additional effort to manage and coordinate new threads and increases the potential for miscommunication between developers. There is also the potential for creating too many child threads (i.e., an exponential explosion), where the benefits gained by decomposing the problem into smaller pieces is outweighed by the resources required to manage the threads.

Because threads incur additional overhead, new threads should only be spawned when the benefits of decomposing the problem being solved into smaller pieces is greater than the cost of managing the child threads. Making this determination is up to the

individual, but some criteria that can be used for determining when to create child threads are when the problem at hand is too complex to be easily visualized/understood by the designer/engineer, when the solution to the problem at hand contains multiple unrelated components which themselves are of substantial size or complexity, or the solution to the problem at hand contains a substantial number of components that may have drastically different life-cycles.

Even though guidelines may be followed for determining when to create new threads, developers can still make poor decisions. Another technique to help guard against the misuse of thread recursion is to require thread managers to have a peer review by other developers before being allowed to create child threads. In addition, developers should simply be educated about the potential of abusing recursion and its consequences. Making them more aware of the potential problems may make them think twice about spawning new threads.

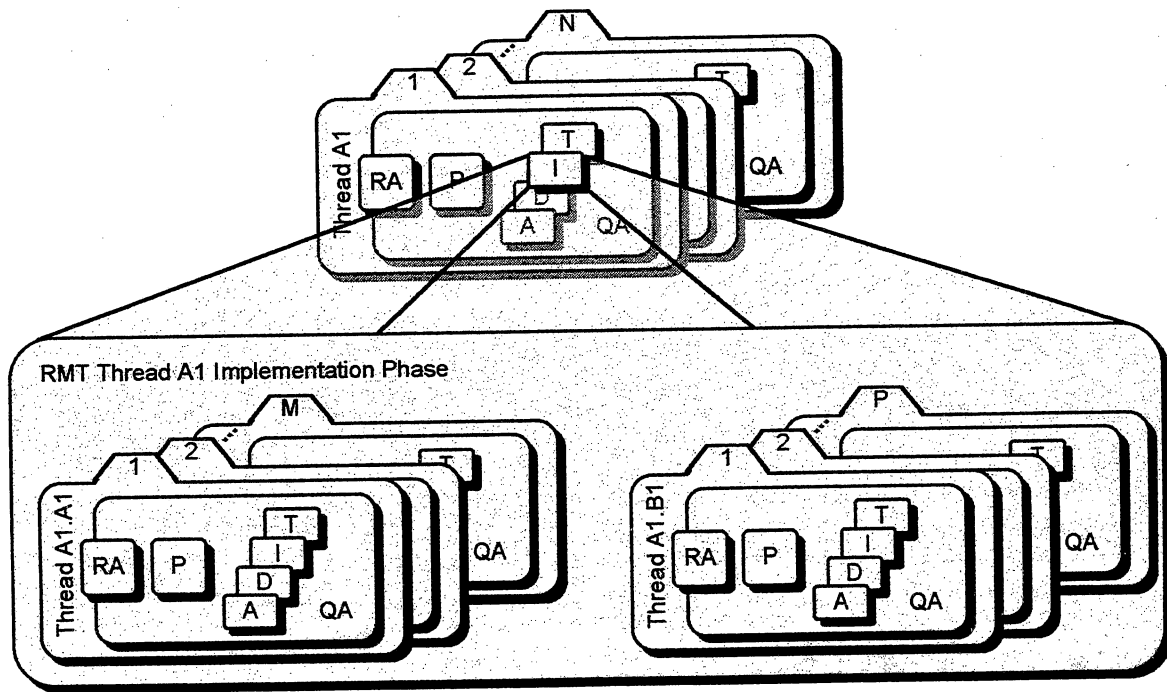


Figure 3.3: RMT Thread with Recursion

3.1.1.3 Reusability

Reusability has long been a goal of software engineering methods. It promises to reduce development costs/effort and improve quality. As mentioned previously, object-oriented technology was originally promoted as offering a higher degree of reusability that it has not been able to deliver. People have realized that both building reusable software components and reusing existing software components is not something that happens automatically as a result of using a certain methodology or technology, but that it is something that is a conscious decision that requires planning and significant effort to successfully employ.

Reusability can take many forms, ranging from high-level design information such as design patterns to low-level source code reuse. Although reuse is simply a tool to

perform an activity, such as a design or implementation, a development life-cycle should make some allowances for it. The following sections present a brief discussion of the forms of reuse that RMT encourages.

3.1.1.3.1 Source Code Reuse

Source code reuse is probably the most efficient and commonly recognized form of reuse. Reusing existing source code provides perhaps the ultimate benefit of software development. It greatly reduces development time and costs, and it improves the quality of the resulting software because the reused components themselves are (or should be) of high quality. Mili [Mili-95] attribute source code reuse as the only technically feasible factor to leverage an order of magnitude improvement in programmer productivity. While source code reuse is, and has been, a highly sought after goal of software development it has not been achieved to the degree hoped as a result of object-oriented, or any other, technologies.

Alfred and Mellor [Alfred-95] believe that one reason wide-scale reuse has not occurred is because the process of reusing software is difficult and time consuming. The design and implementation of reusable classes is much different than classes designed for one-time use. The design and implementation of classes for one-time use tends to be influenced by the system the classes are currently being developed within, and do not take into consideration other issues which affect their ability for reuse. Truly reusable classes need to be more generalized than their single use counter parts. McGregor and Sykes [McGregor-92] believe that to develop reusable software components the life-cycle of class development should be independent from the application life-cycle. The reason for

this is so that the class can be implemented to support a (more) complete description of the entity rather than simply what is needed for the current system. When implementing a new class, this usually involves a fully-defined base class representing the complete entity and a specialized derived class for the current system.

There are also implementation issues with reusing source code. For example, in C++ the decision to declare a member function in a class as virtual or non-virtual can effect the behavior of classes developed by others that inherit from that class. Other implementation difficulties of reusing source code are platform portability and language compatibility. As a result, designing and developing reusable classes involves more effort (and expense) than classes for one-time reuse.

The generation of reusable classes is only half of the problem. Once reusable classes have been created, classes to be reused must be identified during software development in an efficient manner. Reviewing source code manually to locate candidate classes is impractical, so some form of cataloging should be used. The Object Reuse Classification Analyzer (ORCA) and Automated Hypertext Reuse Search Tool (AMHYRST) projects are examples of systems that can be used to support searching repositories of reusable software objects [Isakowitz-96]. Another problem with reusing existing classes is that it is rare that classes can be reused as-is without any modifications. Many times, the effort required to modify the reused class involves more effort than developing the class from scratch.

Regardless of the problems associated with source code reuse, software engineers will continue to pursue source code reuse to their advantage, so a development life-cycle

should accommodate the reuse of existing source code (classes). During the design phase of an RMT thread, the designers should evaluate existing class libraries to determine if there are existing components that implement the given design specification. Software components built as the result of an RMT thread can be integrated into a class library after the thread (component) has been completed.

3.1.1.3.2 Design Patterns

Software systems generally contain recurring patterns of solutions to problems, whether they are real-world problems or software implementation problems. When given a new problem, it would be wasteful to implement a new solution if someone else had already solved it. The ideal situation would be to reuse the existing source code used to solve the problem, but this may not be possible in all cases. The next best situation would be to consult the individual(s) who had already solved the problem, get a description of the solution, and implement it. Many times, however, the individual(s) may not be available, or they may have even forgotten how they solved the problem. The existing source code could be examined and the solution extracted, but this takes valuable time and may result in an incorrect interpretation of the solution. In this case it would be useful for the original designer to document the solution that was implemented (while they still have a detailed knowledge of the design) so that other people could use the same approach when they encounter the same problem. This is what design patterns do. They document the design of a software component that solves a particular problem.

Because of the many difficulties of reusing source code, design patterns are the next logical step for achieving reuse. Another reason design patterns are so significant is

that object modeling is difficult to get correct the first time, and generally involve several iterations [Rumbaugh-91]. Design patterns are intended to be solutions that have been implemented and proven to work. This eliminates the time for others to evolve a design, which may or may not be correct. While a number of methodologists have defined what information is included in a design pattern, most are based on what is called the Alexandrian form [Coplien-94] which draws from the work by the architect Christopher Alexander. The Alexandrian form includes the pattern name, a description of the problem, the context of the pattern, any limitations of the pattern, the solution, examples, outstanding issues, and the rationale behind the solution. Other formats exist for describing design patterns, such as [Gamma-95], which are also based on the Alexandrian form.

During the design and implementation phases of RMT threads, designers/developers should review existing design patterns for solutions to problems identified during these phases.

3.1.1.3.3 Open-Ended Architectures

Once development of a software system has begun, the cost of making changes to system requirements can be significant. Three factors that may determine the cost of changes to system requirements are the size of the change, the time at which the change is introduced, and the architecture of the underlying software implementation. First, the cost of a change is relative to the severity the change; the more significant the change, the greater the cost [Botting-97]. What may seem to be a small change to a user may require significant changes and cost to the developers. Second, the cost of change is relative to the

time at which it is introduced during the development process. The later in the development process that the change is introduced, the costlier the change is [Pressman-97]. Lastly, the underlying software implementation can drastically influence the cost of a change independent of the size of the change and the time that it is introduced. If a software implementation is not malleable, even a small change may require significant modifications to the implementation.

Developers do not have control over the size of a change or the time a change is introduced during development, but they can control how the underlying software is implemented. In Gilb's [Gilb-88] description of the evolutionary delivery method, a critical issue that contributes to the success or failure of a project is open-ended architectures. Because evolutionary development is designed to accommodate change during the evolution of a system, and changes can be costly, the underlying system should be designed and implemented in such a way that changes can be made to the system without incurring significant effort. Open-ended techniques "are quite simply any solution idea which displays strong attributes of adaptability, hereunder extendibility, portability and improvability." [Gilb-88] Table 3.1 summarizes Gilb's basic principles of open-ended design. Because Microsoft uses an iterative life-cycle for developing software, they have adopted the use of similar guidelines for developing their product architectures [Cusumano-95]. Microsoft refers to this as flexible architectures.

| |
|--|
| All solution ideas will to some degree allow change in a measurable way. |
| Each solution idea has multiple ease-of-change attributes. |
| The expected range of each solution idea's ease-of-change attributes can be noted and used to select them for new designs. |
| The need for open-endedness is relative to a particular project's requirements. |
| Each open-ended solution idea has side-effects which must ultimately be the basis for judging the ideas for possible use. |
| You cannot maximize the use of open-endedness—but must always consider the balance of all solution attributes against all requirements. |
| You cannot finally select one particular open-ended design idea without knowing which other design ideas are also going to be included. |
| There is no final set of open-ended design ideas for a system; dynamic change is required and inevitable because of the external environment change. |
| Open-endedness will, by definition, cost less in the long term, but not necessarily more in the short term. |
| If you don't consciously choose an open architecture initially, your system's evolution will teach you about it the hard way. |

Table 3.1: Gilb's Basic Principles of Open-Ended Design

Because of the iterative nature of RMT threads, some initial thread iterations and all subsequent thread iteration are based upon some existing component(s) (in varying levels of completeness). It is possible that initial thread iterations may be based upon some existing component that requires modifications or improvements rather than implementing a software component from scratch. The requirements of an iteration may require modification, deletions, additions, and modifications, to the underlying system. If the underlying system is not designed and implemented to accommodate change (i.e., open-ended), it is likely that a significant effort will be required to alter the underlying system to integrate the modifications for the current thread. Because changes may occur during each

iteration of a thread, the software may require modifications during each iteration, potentially magnifying the cost of implementing these changes.

Developing software is much like Gilb's analogy of a chess game. Your long-term goal is to defeat your opponent, so you could plan a number of moves to carry out that goal. Your opponents moves, however, are not predictable so you have to make contingency plans. The number of possible combinations for your moves and your opponents counter-moves are astronomical, and you can not realistically account for all of them. Therefore, the only move that really counts is the next one. Since change is inevitable, it is more effective to put your energy into being able to respond to your opponents move while still moving toward your objective than to plan in detail exactly what you are going to do. The same is true of software development, and open-ended architectures are one technique for responding to change. Therefore, software developed using RMT should follow the principles of open-ended architectures to reduce the amount of effort to accommodate change.

Open-ended architectures are not without their costs. Much like developing truly reusable software components, they are more difficult to design and implement, and take more time. However, its effectiveness can not be evaluated based upon the initial development cost because it is a long-term investment (just like software reuse). Initially, the cost will seem excessive; spending more time/effort than is required to design/implement the immediate requirements. However, during later iterations when changes and enhancements occur, the time and effort saved because of the flexible architecture can significantly outweigh the initial overhead.

While open-ended architectures do require additional effort to implement, there are some elements of object-oriented technology that make it easier: abstraction, encapsulation, modularity, and hierarchies [Booch-94]. These elements have long been promoted as good software engineering techniques for traditional methods and technologies, but they are an inherent characteristic of the object-oriented approach, making it easier for a developer to design and implement software with open-ended qualities. Using object-oriented technology does not guarantee that the software produced will contain open-ended qualities, poor designers can still make poor designs, but object-oriented technology definitely makes building open-ended architectures easier.

Shaw [Shaw-84] defines abstraction as “a simplified description, or specification, of a system that emphasizes some of the system’s details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary”. Encapsulation is a technique that hides the internal details of an abstraction, or object, from the user of the abstraction. This is usually done by separating the external view of the object, commonly referred to as its interface, from the implementation of the object. Modularity is a technique of organizing a system into a number of cohesive and loosely coupled units, or modules. In compiled programming languages, such as C and C++, a module is simply a source code file that can be compiled separately. While modularity helps divide a system into logically related abstractions, a hierarchy allows a developer to rank and order abstractions. Each of these elements aids in both the

conceptual (i.e., analysis and design) and physical (i.e., implementation) construction of open-ended architectures.

Even though open-ended architectures are really a design and implementation technique, it is of such importance that it affects how the life-cycle process is defined because an evolutionary process can fail horribly if the designed and implemented system is not open-ended. For this reason, users of RMT are encouraged to follow the principles of open-ended architectures.

3.1.2 Benefits of Threads

The purpose of using threads as abstractions of the development process is to provide some form of control or management for a complex process. As a result of using threads as a form of control, they provide a mechanism for monitoring progress during development, allow parallel development, and support multiple levels of abstraction. The following sections discuss these benefits in detail.

3.1.2.1 Monitoring Progress

Perhaps the single greatest benefit of RMT is its ability to monitor progress. RMT supports the task of monitoring progress during development by providing a mechanism that makes the process of evaluating and interpreting progress estimates easier for developers. Rather than requiring developers to estimate progress for all the software components of a system at one level of abstraction, RMT divides this estimation into smaller units of abstraction: iterations and the thread hierarchy. This mechanism still requires developers to make their “best guess” (i.e., estimate), but only for a small unit of abstraction, not for a large system.

Progress estimation begins at the smallest unit of abstraction in RMT, a class. The implementation of a class is performed within the conceptual unit of a thread, which is partitioned into a number of iterations. Before the implementation of the class actually begins, each iteration is assigned a percentage of overall effort required to implement the class (the sum of the percentages for all iterations is 100%). Progress is measured by summing up the assigned percentages of iterations that have been completed, plus the assigned percentage of the current (incomplete) iteration multiplied by its estimated progress. For example, consider the implementation of a class that is partitioned into three iterations with percentages of 40%, 35%, and 25% of the overall implementation effort, respectively, given to each iteration. If the first iteration is completed and the second iteration is 50% completed, the overall implementation is 57.5% complete $((40\% * 1.0) + (35\% * 0.5) + (25\% * 0.0) = 57.5\%)$.

Progress estimates need to be updated frequently, at each iteration, to accommodate any changes that may occur that would affect the original estimates. For example, if new iterations are added the estimated percentages need to be revised to reflect the new set of iterations.

Once the progress of individual threads can be determined, the progress of implementation phases which have spawned child threads can be determined. The progress of an implementation phase is simply the sum of the weighted progress estimates of each of its child threads. In the same fashion that each thread iteration is assigned a percentage of the overall effort for the thread, child threads are assigned a weighted value indicating the percentage of effort of the implementation phase of the parent thread that the child

thread represents. For example, if the implementation phase of a thread has two child threads, A and B, where A constitutes 75% of the implementation effort and B constitutes 25% of the implementation effort, weights of 0.75 and 0.25 will be assigned to each of the child threads, respectively. If thread A is 25% complete and thread B is 75% complete, the overall progress of the parent threads implementation phase is 37.5% $((0.75 * 0.25) + (0.25 * 0.75) = 0.1875 + 0.1875 = 0.375)$

While this still requires the developers to estimate the percentage of overall effort that each iteration and child thread represent, it does provide some systematic method for estimating progress of complex components and an entire system.

3.1.2.2 Multiple Abstraction Levels

When applying RMT to a particular project, all of the threads are organized in a hierarchy. Each level in the thread hierarchy represents a different level of abstraction. High-level threads address general overall system requirements while low-level threads address the requirements for individual classes. Each thread abstraction level is usually managed and implemented by different developers because each abstraction level requires a different skill set and expertise. While there can be any number of abstraction levels in a particular project, there are three broad classifications: project-level, subsystem-level, and class-level.

Threads in the project-level category address the high-level (broad) system requirements. The highest level thread is the root thread, which represents the entire system being developed. All other threads are spawned from the root thread. Brooks believes that the project architect “is responsible for the conceptual integrity of all aspects

of the product perceivable by the user” and represents the interests of the user during the system development. [Brooks-95] It is the project architect who should be responsible for the management of the root thread. Brooks also feels that the project architect is responsible for partitioning the overall system into subsystems. Each of these subsystems will have its own architect, which may or may not be the project architect. Class-level threads represent the threads that deal with the lowest level of detail (the most specific), which is the actual implementation of a class. Software engineers and programmers are responsible for class-level threads. Subsystem-level threads represent the intermediate threads between the project-level and class-level threads, which deal with subsystems and modules. Project designers are generally responsible for subsystem-level threads, although the project architect may be involved for higher-level subsystem threads and software engineers may be involved for lower-level subsystem threads, depending upon the availability of resources.

How the development staff are organized can influence the quality and timeliness of software development. Poorly organized teams can have very undesirable effects on development, making communication between developers difficult or unreliable, introducing delays, etc. Each thread has a set of assigned team members, and by structuring threads as a hierarchy RMT provides an organization to the development team. Because each team member of a thread has well-defined responsibilities, the hierarchy of RMT threads provides well-known points of communication throughout the entire development team so individuals can identify who to contact when there is a question or problem related to a particular component or thread.

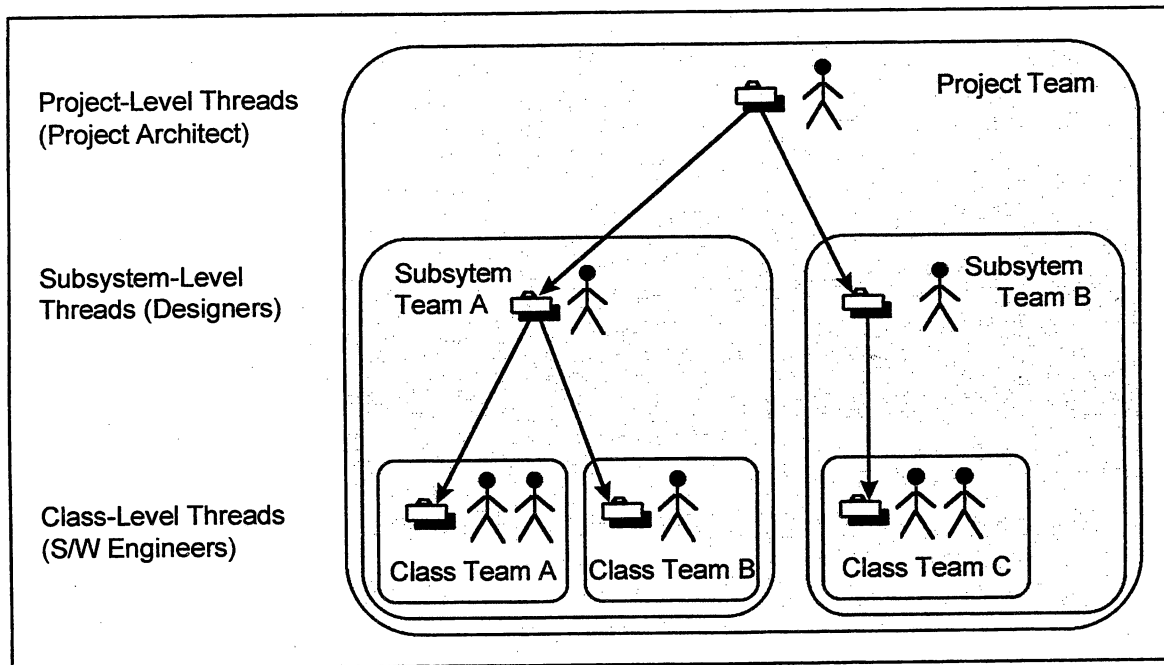


Figure 3.4: Levels of Thread Abstractions and Thread Managers

A thread manager is responsible for implementing the requirements assigned to their thread. Because this thread may spawn other child threads, the manager is also responsible for these child threads. The requirements given to a child thread is essentially a contract between the managers of the parent and child threads for what the child thread needs to do. This clearly defines the responsibility of each individual in the development process. In addition, the thread manager is responsible for notifying the manager of the parent thread when their thread is completed.

The hierarchy of threads can have its disadvantages. First, each new child thread involves the additional overhead of a person to manage the thread. The addition of new threads can also reduce the conceptual integrity of the project because as the high-level requirements “trickle” down through the thread hierarchy, the essence of the requirements may be lost or fade because they have been decomposed into many independent pieces

(the “can’t see the forest through the trees” syndrome). The thread hierarchy also adds the potential for miscommunication simply because there are more individuals introduced in the development chain from the user to the engineer who implements the software. Because of these potential problems, new threads should be created only after careful consideration. Section 3.1.1.2 discusses some guidelines for when to create new threads. In addition, the requirements that are passed to child threads should be as close to the original thread requirements as possible so that the conceptual integrity of the system is maintained.

Within each thread, the distribution of development effort for each phase depends upon the level of abstraction. Project-level threads generally involve more effort in the planning and requirements analysis phases, subsystem-level threads involve more analysis and design activities, and class-level threads involves more implementation.

3.1.2.3 Parallel Development

With the overlap between the analysis, design, implementation, testing, and quality assurance phases and the recursive application of threads, parallel development is introduced. Parallel development simply means that there may be more than one activity being performed at any given time. In sequential life-cycles, parallel development is impossible because the development effort is required to be in a single phase at any given time. This results in the inefficient use of resources because team members specializing in different areas may be idle while others are not, and vice versa.

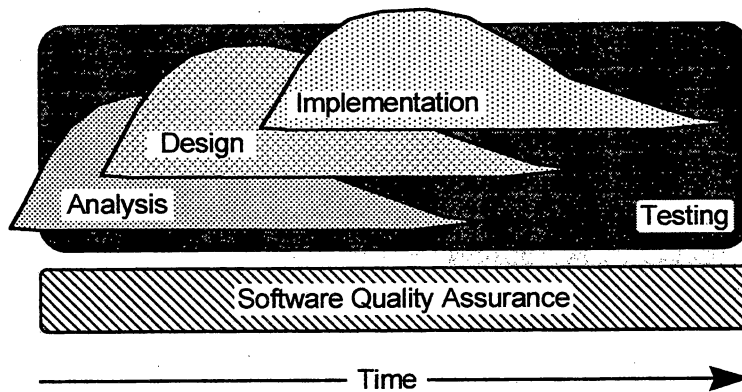


Figure 3.5: Distribution of Activities, in Practice, of Traditional Sequential Software Life-Cycles [Berard-93]

Berard [Berard-93] describes that in practice, even in traditional sequential life-cycle models there is a great deal of overlap between phases (see figure 3.5). Even though there is overlap of activities, in many life-cycles a majority of the planning/requirements activities happen early, implementation happens in the middle, and testing happens at the end of the life-cycle. In RMT, a thread may be in multiple phases simultaneously and there may be any number of threads executing at any given time during a development cycle, each of which may be executing at a different level of abstraction; therefore, there is a high-degree of parallel development. As a result of the high-degree of parallel development, there is a very efficient use of resources (developers). At a particular moment during the development cycle project architects may be analyzing high-level requirements for one subsystem, project designers may be designing other subsystems, software engineers may be implementing other components, and quality assurance specialists may be testing other components all at the same time. At the beginning and end of high-level threads, such as the root thread, there will tend to be some team members

performing a majority of the work while others will have very little to do. This can not be avoided unless team members are qualified to perform different activities of the development life-cycle (although, there are still only so many people that can have their hands in the cookie jar at the same time).

Within programming languages, a mechanism is usually provided to synchronize the parallel execution of multiple threads so that they can coordinate their activities to avoid undesired side-effects (e.g., concurrent access/modification of data). Within RMT, development threads may also need to be synchronized with other threads, although the reasons are different than those of programming languages. RMT threads need to synchronize with other threads so that the software components being implemented by one thread will work with software components being developed by other threads. Thread synchronization occurs when all of the phases within a thread iteration have been completed. This is an important concept because it implies that any child threads that may have been spawned during the implementation phase have been terminated and the software component satisfies the requirements of the thread iteration (i.e., it satisfies its contract).

In order for parent threads to know when child threads have been synchronized, they must be able to communicate. This is done by the manager of a thread who reports to the manager of the parent thread that the child thread is completed (this is discussed in more detail in the following section). Thread synchronization occurs at all levels of abstractions, but is most significant at the root thread which represents the entire software system. The synchronization of thread iterations at the root thread implies that the

incremental version of the entire system is complete and functional. This may involve a delivery of the system to the user or internal project teams for evaluation and feedback.

The concepts of iteration and synchronization within RMT is very similar to that of milestones. A milestone is simply an event at which time a number of objectives are to be completed. Milestones usually have an associated estimated or required completion date and can represent deadlines for user deliverables, an indicator when certain objectives have been completed, or internal goals identifying the completion of a particular component. Within RMT, the synchronization (completion) of a thread is synonymous with a milestone. If the completion of a particular thread iteration is deemed significant, a milestone may be established at the end of that thread iteration. Microsoft uses a development life-cycle which divides large projects into three to four major milestone product releases [Cusumano-95].

Another benefit of thread synchronization (and iterative development) is that, if the implementation has been done correctly and diligently, at each point in the development process there is a working, tested (but incomplete) version of the system that could theoretically be shipped to the user. Microsoft uses an incremental technique called the synch-and-stabilize process, which uses frequent “builds” (synchronization) and stabilization periods of the system to facilitate this. [Cusumano-95] The synchronization part of the process involves the “daily build and smoke test” [Cusumano-95, McConnell-96]. The daily build involves the compilation and linking of all source code into executable programs each day. If the build fails, fixing the build becomes the highest priority. Once the build is successful, the “smoke test” is run to verify that there are no major problems

with the system. It is not a comprehensive set of tests, but tests the major components of the software to prevent quality from degrading and integration problems from becoming significant.

3.2 RMT Activities/Phases

An RMT thread is divided up into a number of phases that carry out different portions of the development process. Each phase has a well-defined goal with specific inputs and outputs and may involve a number of tasks to carry out these goals. Generally, the results of one phase are inputs for the next phase. These phases are not unique to RMT, and in fact are common to many software life-cycles. While the phases are undertaken in a sequential order, there may be overlap between phases, especially with the quality assurance phase which happens simultaneously with all phases of thread but culminates at the end of the thread.

Most of the RMT thread phases produce documentation (e.g., textual and graphical) as output. This documentation is critical to applying RMT effectively because the documentation not only provides developers with a clear and concise description of existing components, but it represents the state of a thread. Because many developers may work on multiple threads, it is possible that some threads may “go to sleep” temporarily because no one is available to work on that thread. At some time later when the developers become available to resume work on these threads, the developers need to continue where they left thread development. The documentation can contain the information describing what state thread development was in when it “went to sleep”.

Since these documents can, and probably will, undergo changes during thread iterations, some method should be employed to maintain a history, or versions, of each of these documents. This allows developers to consult and compare previous versions of the documents. If the client wants to know why the project schedule is behind, maintaining versions of the requirements document may show that a significant number of system requirements were added since the initial iteration.

The following sections describe each of the RMT thread phases. Many object-oriented methodologies have very detailed definitions of what is done (and how) during of these phases. Because RMT does not require the use of a particular methodology, the descriptions present the goals of each phase without specifying the details of how the tasks are performed.

3.2.1 Requirements Analysis

The first phase of any RMT thread is requirements analysis. The goal of this phase is to solicit, analyze, and define the requirements for some software component. These requirements represent a contract between the thread and the client of the thread. Project-level threads generally require the user to provide the developers with an initial set of requirements. The initial set of requirements may be incomplete, inaccurate, inconsistent, vague, or unnecessary. The developers need to improve these requirements to ensure that the requirements are what the user really needs, detail any vague requirements, identify any inconsistencies between requirements, identify any requirements that were not identified, and eliminate unnecessary requirements. This usually involves interviews between the developers and users. The requirements for lower-level threads, subsystem-

level and class-level threads, are usually specified by the developers themselves as part of the system implementation to satisfy higher-level user requirements. Refinement of these requirements generally involves discussion between developers to insure that the requirements are accurate and complete.

3.2.2 Planning

The planning phase takes a set of well defined requirements as input and produces a development plan for the thread. Since the input is a set of well-defined requirements, the planning phase can only begin after the requirements analysis phase has been completed. The primary goals of the planning phase is to estimate the number of iterations required to implement the given set of requirements, prioritize the set of given requirements, and assign each of the requirements to a particular thread iteration. While all of the requirements will be passed to the next phase, only those requirements that are assigned for the current thread iteration are scheduled for implementation. The other requirements are included only for evaluation to avoid any conflicts or dependencies with previous or future thread iterations.

The requirements can be assigned to thread iterations using any method deemed necessary by the project manager, but Gilb [Gilb-88] suggests that requirements should be ranked and prioritized according to the value for the user and the amount of effort required to implement these requirements. Requirements with the larger value to cost ratio should be assigned to early iterations. The development plan should include a specification for each iteration which includes the set of requirements to be addressed in that iteration, the estimated amount of effort required to carry out the iteration, the estimated/required

completion time for the iteration, and the allocation of available resources needed to carry out the thread iteration.

Each organization has its own technique for estimating development effort and scheduling projects, but when scheduling the estimated completion date for thread iterations (a.k.a. milestones) it is suggested that some form of buffering be incorporated into the estimated schedule. Microsoft incorporates some amount of buffering in each major product development milestone to accommodate uncertainties that arise during development to more accurately meet estimated dates [Cusumano-95]. These uncertainties may include scheduling overruns because of misunderstandings of requirements or technical issues, unscheduled requirements, or other unexpected problems. This buffer time should not be used for anticipated tasks such as feature development or testing. In application products, Microsoft usually allocates 20 to 30 percent of the schedule to buffer time [Cusumano-95].

Another critical goal of the planning phase is to produce what Microsoft calls a vision statement [Cusumano-95] and Schach calls a specification document [Schach-96]. This document is based upon the set of system requirements, produced in the previous phase, and specifies precisely what the resulting system is, what functionality it will contain, and any system constraints. In addition to specifying what the product is, the vision statement specifies what the system is not. This is equally important as specifying what the system is. Schach views this document as a contract between the developers and the users as to what constitutes the acceptable criteria for the resulting system.

3.2.3 Analysis

Once the development plan has been completed, analysis of the given requirements can begin, which is the first step of the actual system implementation. The input to this phase is the set of requirements, a development plan, and a risk analysis report for the current thread iteration. The risk analysis report is a result of the quality assurance activity, which is discussed in a later section. The goal of this phase is to fully understand and define the problem to be solved for the given set of requirements. The output of this phase is a clear understanding and definition of the problem, which may take the form of documents and/or diagrams, depending upon the particular methodology being used. This document is called the problem specification. The problem specification will likely include, in addition to a description of the problem, a description of a number of objects/classes (i.e., their name, attributes, and behavior) that were identified during the analysis phase that are problem-specific. These objects or classes may or may not be coded during the subsequent implementation phase, depending upon their relevance in the design and implementation phases. It is possible that an object/class identified during the analysis phase is simply used to describe and model the problem but have no representation in the resulting software.

The requirements scheduled for implementation during the current thread iteration are the primary focus during this phase. Related, or potentially related, requirements may also be considered for analysis during this phase because they may affect the requirements scheduled for implementation during the current thread. Requirements scheduled for implementation during the current thread iteration may not have been implemented yet or

they may be existing requirements that have been implemented during a previous thread iteration but have been modified. If a requirement is new, then it must be analyzed and a new problem specification must be constructed. If the requirements is an existing requirement that has been modified then the previous problem specification for the modified requirement should be compared with the modified requirement of the current thread to identify incompatibilities. A new problem specification should be created for the modified requirement which accounts for the requirement changes. These problem specifications are used as input for the design phase.

During subsequent thread iterations, the software components implemented by previous iterations should be consulted during the analysis of current requirements. This may identify similarities or conflicts with the existing software. Similarities may result in the reuse of design information and/or source code. Conflicts may result in modifications to the existing software to accommodate changes required for the current requirements.

Because many of the input requirements may be unrelated to each other and can be analyzed and specified independently, the specification for some problem areas may be completed before others. Once enough specification information exists for a particular problem area the design phase for the specified problem area can begin. For problem areas that are closely related or dependent on each other, the design of those problems should be delayed until all related problems have been fully analyzed and specified because each specification could change due to later analysis of related problems. Because design activities may begin simultaneously with analysis, the boundary between analysis and design activities is vague. To further cloud the boundary between analysis and design, the

identification and description of analysis objects/classes during the analysis phase may be considered the beginning of the design phase. This is due to the fact that the description of a class is common to all development phases and the initial specification of a class' attributes and behavior begins during the analysis phase.

3.2.4 Design

The goal of the design phase is to specify, in detail, how the underlying software components are to be implemented. The input to this phase is the problem specification document which is a detailed definition of a problem. The output of the design phase is an implementation plan, or design document, which provides a detailed specification of the software component(s) to be implemented, and may take the form of textual documents and/or diagrams, depending upon the design methodology used.

On the initial thread iteration, the design phase involves reviewing the problem specification and constructing the specification (or design) for the software component(s) to solve the specified problem(s). During subsequent thread iterations, the design documentation and source code for the existing system (implemented during previous thread iterations or by other threads) may need to be reviewed to identify any impact that the problem specification of the current thread iteration will have on the existing system. Commonly, each thread iteration will require new functionality to be added to existing components, which may require modification to the existing component.

As a solution to the problem specification input from the analysis phase is outlined during the design phase, new objects/classes that were not identified in the problem

specification may be introduced in the design document to fully-specify the solution to the problem [McGregor- 92]. These objects are hidden from the user.

The design for a software component may begin when enough analysis information exists to fully specify the problem the component needs to solve. Similar to the overlap between the analysis and design phases, the implementation phase may begin before the design phase is completed. Since a large number of subsystems, modules, and classes may be identified and specified during the design phase, the design for some components may be completed before others. In this situation, the actual implementation for these components may begin before all design activities have been completed. It would be prudent only to begin implementing components that are either independent of other components, or related to components which have complete design information.

If libraries of design patterns are available to the designer, they should be evaluated during the design phase to determine if there are existing designs that are applicable to the problem at hand. If applicable patterns are located, then the existing design information should be reused and incorporated into the design specification. If libraries of reusable software components are available, they should also be consulted to determine if there are existing software components or frameworks that could be used during the implementation of the design specification. This is done because the design specification of the potential component(s) can be made to conform to the existing software component(s), if the integrity of the design is not compromised. Then the existing components could be reused with little or no software modifications.

3.2.5 Implementation

The implementation phase is where the actual software coding occurs. The input to this phase is a design specification for a particular software component. This component may be a single class or an entire software system. The output of this phase is a fully implemented software component that adheres to the given design specification. Since the design specification generated during the design phase may contain specifications for a number of independent software components, it is possible for implementation to begin before the design specification for all software components has been completed. The implementation of a software component may begin when the design specification for that component has been completed in the design phase.

If the design specification is for a low-level software component (e.g., a class), then the component is coded according to the given design specification. If the given design specification is for a high-level software component (e.g., a module or subsystem), then the design specification is decomposed into a number of smaller, cohesive pieces, and new threads are spawned to implement each piece. To implement the given design specification, new classes may be identified that were not specified in the design specification but are required for implementing the design (see [McGregor92]).

To promote software reuse, a design pattern library and source code library should be reviewed, if available, for compatible designs and/or source code before implementing new components. If compatible design information or existing components are located, they should be reused appropriately. This improves development time and software quality.

On the initial thread iteration when there is no existing software component, the source code to be implemented will be done from scratch. On initial iterations where a software component already exists or during subsequent thread iterations, it is possible that the source code to be implemented during this phase will need to be integrated into some existing version of the overall system. If this is the case, then the design specifications and source code of the existing system affected by and/or related to the changes outlined in the design specification should be reviewed prior to coding. This review is done to reduce (and hopefully eliminate) potential problems during or after the required coding. If new components are coded, then the review involves understanding how the existing system and the new software component will interact. If any existing components require modification, the review involves the identification of any behavioral changes to existing methods and any “client” components invoking these methods.

3.2.6 Quality Assurance

Quality assurance is a broad term which means involves many different activities at many different times during software development. The quality assurance phase encompasses all activities required to ensure the quality of the software produced. It occurs simultaneously with all other thread phases, but culminates after the implementation phase.

The most common form of quality assurance is testing. The type of testing performed during a thread depends upon the abstraction level of the thread. Unit testing occurs during class-level threads; integration testing occurs at system-level threads; and functional testing occurs during the project-level thread(s). As with the development of

the system source code, the development of test cases should involve reviewing existing design patterns and source code libraries, reusing designs and/or components for test cases where possible.

It is generally preferred that the person performing testing (design, implementation, and execution) is not the same person who developed the software being tested. This is because the developer's view of the software is tainted with implementation details, where an independent test engineer is removed from the implementation details and is more concerned with behavior. This also allows for a higher degree of parallel development which results in a shorter development time. Microsoft, for example, tries to pair up a test engineer with each developer [Cusumano-95].

Development of test cases can begin as early as the analysis phase. A high-level design for test cases can begin as soon as enough stable design information exists for a component, which may occur before the design phase has been completed. It is wise to only begin a test case design when the system component design is relatively stable and is not likely to change drastically. When the design specification for a system component has been completed and the implementation phase begins (note that the design phase may not yet be completed), the complete test case design can be begin. The final test case design may vary greatly in formality and detail depending on the complexity of the test and available resources (e.g., time, budget, etc.). The actual implementation of the test case may begin once the test case design has been completed. This may occur during the implementation phase before the component to be tested has been completed. Allowing for development of test cases to happen in parallel with the system development

streamlines the development process. If personnel are not available during the design and implementation phases to develop the test cases, the test cases can be developed following the implementation phase.

Testing, however, is not the only task of the quality assurance phase. It may also involve risk management, verifying that the software meets all of the system requirements, and assuring consistency of information between the analysis, design, and implementation phases.

3.2.6.1 Risk Management

While not part of the traditional quality assurance activities, risk management is another task performed during the quality assurance phase, primarily because, like quality assurance, it occurs simultaneously during all other thread activities. As mentioned briefly in chapter one, risk management is comprised of three distinct tasks: risk analysis, risk monitoring and mitigation, and risk resolution.

3.2.6.1.1 Risk Analysis

When dealing with problems during software development it is better to prepare for potential problems rather than reacting to them after they happen. That is what risk analysis is intended to be, a proactive strategy for dealing with problems during software development. The “risk analysis” task focuses on the identification, evaluation, and planning of potential risks associated with developing software.

The first goal of this task is to identify any potential threats to the development of a thread based upon the given requirements, scheduling requirements, development environment (e.g., personnel, technology, etc.), existing systems, and any other identified

factors. There are a number of methods and techniques that can be followed for identifying risks that identify many different types of risks. Pressman [Pressman-97] suggests the use of a risk item checklist of questions that can be used to identify risks. This checklist is divided into several sub-categories of known and predictable risks:

- “Product size—risks associated with the overall size of the software to be built or modified
- Business impact—risks associated with constraints imposed by management or the marketplace
- Customer characteristics—risks associated with the sophistication of the customer and the developer’s ability to communicate with the customer in a timely manner
- Process definition—risks associated with the degree to which the software process has been defined and is followed by the development organization
- Development environment—risks associated with the availability and quality of the tools to be used to build the product
- Technology to be built—risks associated with the complexity of the system to be built and the ‘newness’ of the technology that is packaged by the system
- Staff size and experience—risks associated with the overall technical and project experience of the software engineers who will do the work.” [Pressman-97]

While there are other techniques for identifying risks, much of the risk identification can be attributed to the skill and experience of the individual.

The second goal of this task is to rate the potential costs of each identified risk. There are a number of methods to do this, but most involve some classification based upon the probability that the risk will occur and the impact or consequences as a result of the identified risk. The probability of each risk occurring is specified as a percentage, while the impact or consequence of each risk is some scalar value assigned by the developer. An overall risk factor can be calculated for each identified risk by multiplying the probability by the impact. The identified risks can then be sorted and the highest probability-to-impact

value should receive the most attention. Pressman [Pressman-97] and Jacobson [Jacobson-92] suggest the use of a risk table sorted by probability and impact/consequence. Low order risks may be deemed not significant enough to warrant further consideration. Pressman refers to this as drawing a cut-off line for all identified risks less than some factor.

Once risks have been identified and classified, each risk above the cut-off line should be further classified as severe, moderate, or mild risks. Development for the current thread may continue in one of three ways based upon these sub-classifications. Mild risks have a lower probability-to-impact value so thread development can continue as normal but an avoidance and contingency plan is made to manage the risk should it occur. Moderate risks have enough significance to temporarily put the thread development on hold until some risk management/resolution technique (see section 3.2.6.1.3) can either reduce the priority of the risk, eliminate the risk entirely, or promote the risk to a severe risk. A severe risk is one that is considered to have potential risks so great that the current thread must be terminated. This is similar to the spiral model [Boehm-88] in that at the beginning of each spiral (and in the case of RMT, a thread) some risk resolution technique may be used to evaluate unknown risks before continuing development.

While the risk analysis task is not a independent phase defined within an RMT thread, it begins during the planning phase and must be completed before the analysis phase can begin. It is not required that the planning phase be completed prior to beginning risk analysis, but it would be prudent only to analyze risks based upon requirements and development plans that are fairly complete and accurate. Analyzing potential risks for

requirements and development plans that will only change later will result in wasted effort. When beginning risk analysis, a fully specified set of software component requirements and a development plan is required (which may be complete for the related component(s), but not for the entire thread). The result of this task is a risk analysis report which includes a list of potential risks, the probability and impact of each risk, any information gained during prototype threads used to clarify risks, and contingency plans for each risk.

3.2.6.1.2 Risk Monitoring and Avoidance

The risk monitoring and avoidance tasks occur during all thread phases following the planning phase. Risk monitoring involves identifying whether the probability that any of the identified risks has increased or decreased based upon a number of factors. Should the probability of a risk increase enough so that the probability-to-impact value becomes significant, risk avoidance techniques may be employed to decrease the significance of the risk, or to eliminate it entirely. If the probability that a risk will occur becomes so great and can not be avoided, it is possible that the current thread development may stop until the risk is either resolved using the methods mentioned in the previous section or the thread is immediately terminated.

3.2.6.1.3 Risk Resolution

The risk resolution task is the action taken when a risk has either occurred or is categorized as being significant enough to put development temporarily on hold. If the risk has not yet occurred but has been deemed significant enough to stop development, then some proof needs to be shown that either reduces the priority of the risk or that the risk may be addressed using some development or implementation technique. To do this a

prototype, benchmark, or proof-of-concept implementation may be used to clarify and/or resolve the identified risk, which involves the spawning of a new thread with requirements to address these issues. After the “prototype” thread has been completed either (1) another prototype thread will be spawned if the previous prototype was inconclusive, (2) the risk will be downgraded to a lower priority as a result of the information gained during the prototype thread and the current thread development can continue, or (3) the identified risks will be promoted to a severe risk and the thread will be immediately terminated. If the risk has already occurred, then the contingency plan, which was created during the risk analysis task, needs to be implemented.

It is important to note that while the estimated effort for risk analysis can be reasonably estimated in the development plan, risk management may incur additional overhead to the thread. Should such events occur, the development plan should be modified and re-evaluated accordingly.

3.2.6.1.4 RMT Risks

As part of the risk management activities, there are certain risks inherent to RMT that can have negative results on the development process that should be identified and monitored during development. These risks are the misuse of iteration and recursion, making incorrect progress estimates, and miscommunication between team members.

As mentioned previously, the misuse of iteration and/or recursion during development can have detrimental effects on development. Therefore, the use of iteration and recursion should be carefully monitored to prevent developers from abusing these techniques. If a thread manager notices that a developer is using what seems to be an

excessive number of iterations and/or child threads, the manager should inquire about the reasons for using the iterations and/or child threads. If they are deemed excessive, then the developers need to remove the extra iterations/child threads and adjust the appropriate development plans accordingly.

Another potential risk is making inaccurate estimates for determining progress of threads. When estimating the amount of effort required for a particular thread iteration or child thread, developers can make mistakes. If, at any point during development, an estimate is discovered to be incorrect, the development plan for that thread should be updated accordingly and any parent thread should be notified of the changes. Any impact that such changes have on the schedule should also be made to the development plan, and the parent should be notified. These modifications to the development plan is very similar to what happens at the beginning of each thread iteration when the thread requirements change.

Finally, because development is organized as a hierarchy, there is a potential for miscommunication between team members simply because there are more individuals involved in the chain from user requirements (top-level) to actual class implementation (bottom-level) and there are more teams working independently of each other. A possible risk avoidance technique for this problem is to have regular meetings with developers from different teams to review the progress and direction of each team. Also, thread managers should meet with the managers of parent and child threads to review thread requirements, progress, directions, and to voice any assumptions that any teams may have of other

teams. These techniques can help surface any problems that may occur due to miscommunication between team members.

3.2.6.2 Traceability

Berard defines traceability as “the degree of ease with which a concept, idea, or other item may be followed from one point in a process to either a succeeding, or preceding, point in the same process.” [Berard-93] Within RMT traceability means that each of the original system requirements can be traced to the resulting system. This allows the developers, specifically the quality assurance engineers, to verify that resulting system satisfies the original requirements. To facilitate traceability, each requirement (and/or element in the vision statement) should be named or numbered and referenced in a test plan to verify that the requirement has been met. For the root thread the test plan should contain references to the overall system requirements. For threads other than the root, each requirement input to that thread should be referenced and/or verified in the testing phase.

Rather than simply tracing requirements from the requirements analysis and planning phases directly to the testing phase, which may be difficult and or time consuming, tracing requirements should be performed during intermediate development activities. Specifically, every specification made in the vision statement (which is not an exclusion specification) should be traceable to a requirement in the requirements specification and every aspect of the design document can be traced to the vision statement [Schach-96]. This occurs during the planning phase, when the vision statement is being prepared, and the design phase, when the design document is prepared. Both the

vision statement and design document must satisfy this criteria before being considered complete.

3.3 Documentation

As discussed in the previous sections, each thread phase takes some input and produces some form of output. Most phases produce some form of both textual and graphical documentation, with the exception of the implementation phase which produces source code. The following table summarizes the documentation generated by each thread phase.

| RMT Thread Phase | Resulting Documentation |
|-----------------------|------------------------------------|
| Requirements Analysis | Requirements document |
| Planning | Development plan, Vision statement |
| Analysis | Problem specification document |
| Design | Design document |
| Implementation | Source code |
| Quality Assurance | Risk analysis report, test plan |

Table 3.2: Documentation Generated During Thread Phases

Chapter Four-Applying RMT

While the previous chapter presented the concepts and definitions of the components of RMT, this chapter provides a scenario of how to apply RMT to a particular project. While the sample project is intended to illustrate how RMT can be used in practice, many of the details have been left out, such as the actual design specifications. Within the description of each thread iteration only the significant differences from previous iterations will be discussed.

4.1 The Project

The hypothetical project used will be a client/server application to query, insert, and update a database in a multi-user environment. The client portion of the system will be an application with a graphical user interface (GUI) that allows a user to query, display, insert, and update data in a relational database management system (RDBMS) running on a remote server machine on a local area network (LAN). A single server application will communicate with a number of client applications across the network and interface directly with the RDBMS. The server application acts as the liaison between the client application and the RDBMS.

4.1.1 Thread Naming Convention

Because of the iterative and recursive nature of RMT threads, there may be a large number of threads that need to be managed and monitored during development. An explicit hierarchy of threads helps organize development, but it still may become difficult to identify and trace the ancestors and descendants of these threads. In order to quickly

identify the location of a thread and/or thread iteration in the hierarchy of project threads, the following thread naming convention will be used.

There are three elements of a thread iteration that identifies its position within the project thread hierarchy; its parent thread, the iteration number, and its sibling threads.

First, to uniquely identify a thread from its siblings, a thread name contains one or more letters (e.g., A, B, AA, etc.). Secondly, to identify individual thread iterations, a thread name contains a version number which identifies a particular thread iteration (e.g., 1, 2, etc.). Lastly, to identify a threads lineage, a thread name is prefixed with the name of its parent thread followed by a period (.). The root thread has no parent thread so the prefix is omitted. For example, the second iteration of a root thread is named A2, with two child threads named A2.A_n and A2.B_n.

There are a number of tools that are commonly used during software development to maintain a history of source code and documents (e.g., SCCS, RCS). These tools generally use some form of version numbers to identify distinct copies of files. The version numbers of threads could be assigned based upon the version numbering scheme used by such tools so that the thread version numbers would correlate to the version numbers assigned to the actual source code files, design documents, etc.

While this naming convention uniquely identifies a thread in the thread hierarchy, it is not very meaningful to developers. Therefore, an additional name may also be used in conjunction with the unique name.

4.2 The First Iteration

The first thread iteration of an RMT thread is unique from all other thread iterations. The difference is that at the initial thread iteration there is less existing design information or source code so more must be done from scratch. Subsequent thread iterations, however, usually build upon some existing component(s) from previous iterations (unless the existing component is discarded) and developers must take existing designs and source code into consideration. The unique name of the first iteration of the root thread for this example is A1.

During the requirements analysis phase, the following system requirements are identified during user/developer meetings and interviews:

- (1) The client application must communicate with a single server application across a LAN.
- (2) The server application must communicate with a number of client applications across a LAN.
- (3) The server application must interface with an RDBMS.
- (4) The client application must provide a GUI for the user to query data in the RDBMS through the server application (this is the most common operation performed by users).
- (5) The client application must provide a GUI for the user to insert data into the RDBMS (via the server application).

The above system requirements are then formalized into the requirements document, which serves as the basis for project development from this point forward. In reality, there should be many more system requirements, such as hardware/software

specifications, the RDBMS to be used, specific GUI requirements, etc., but were left out for the sake of brevity for this example.

The first task of the planning phase is to prioritize the requirements in order of importance to the user. From the above list, requirements (1) through (3) can be grouped together into a single requirement because they represent the underlying architecture of the entire system. Even though the implementation of this architecture will not be anything that the user will see, it should be given the highest priority. Requirement (4) should be given the next highest ranking of importance. This requirement encompasses both the GUI design and the most commonly used operation of the system. The last requirement, (5), is given lowest priority.

The next task of the planning phase is to estimate the amount of effort required to implement each of these requirements. In order to determine these estimates, developers are consulted for their input. After the estimates are compiled, the developers estimate the number of iterations that will be required to implement the requirements. For this set of requirements, three iterations will be used for the root thread. The first iteration will be an internal milestone that will not be delivered to the user. Requirements (1), (2), and (3) will be addressed in this iteration and will consist of a basic client (with no GUI), server, and RDBMS applications. The client application will be able to connect to the server application, the server application will be able to connect to the RDBMS, and the client and server applications will be able to send and receive dummy requests and responses to simulate normal operations. The second and third iterations will address requirements (4) and (5), respectively, and will both be delivered to the user. At the second iteration, the

GUI for the client application will be in place and the user will be able to perform basic queries against the RDBMS. The third iteration will allow the user to insert new data into the RDBMS. Each of the identified thread iterations will be given an estimated start and end date.

Each iteration for thread A is then assigned some percentage of the overall effort required to implement the entire thread. Iteration A1 represents 40% of the overall effort, iteration A2 represents 25%, and iteration A3 represents 35%.

At some point during the requirements analysis and planning phase, risk analysis begins. This produces a risk analysis report, which must be completed before the analysis phase begins.

Even though the requirement analysis and planning phases address all of the user requirements, only the requirements that are scheduled for the first iteration are considered during the analysis phase; requirements (1), (2), and (3). Four major modules or subsystems are identified as part of the problem: the client application, the server application, a communication subsystem, and a database abstraction subsystem. The specific requirements and behavior for each of these subsystems are analyzed and specified in the problem specification document. In this example, the specifications for the communication subsystem are completed prior to the other subsystems, so the design phase for the communication subsystem actually begins before the problem specification for the remaining subsystems is completed.

As the design phase of the communication subsystem begins, the developers identify a number of classes in a class library that contain the functionality required of the

problem statement, and can be reused in the implementation phase. The existing design documentation for the classes is then incorporated into the design specification for the communication layer.

While the design for the communication subsystem is in progress, the analysis phase is completed and the problem statements for the remaining subsystems were made available. During the design of the remaining subsystems, the developers are unable to identify any existing design patterns or source code for the remaining problem specifications, so the subsystems must be constructed from scratch.

The implementation phase begins with the given design of the four subsystems. Existing components for the communication subsystem have already been identified as solutions for the design, so no implementation is required. The implementation of the client, server, and database subsystems is performed by creating three new child threads, A1.A, A1.B, and A1.C. Thread A1.A implements the client subsystem and contains four iterations (i.e., threads A1.A1, A1.A2, A1.A3, and A1.A4). Thread A1.B implements the server subsystem and contains three iterations (i.e., thread A1.B1, A1.B2, and A1.B3). Thread A1.C implements the database abstraction subsystem and contains three iterations (i.e., A1.C1, A1.C2, and A1.C3). The implementation phase of thread A1 is not completed until threads A1.A, A1.B, and A1.C have been completed.

To allow the manager of thread A1 to monitor the progress of the implementation phase of thread A1, threads A1.A, A1.B, and A1.C are each assigned a percentage of the overall effort of the implementation phase. Thread A1.A represents 55% of the overall implementation effort, thread A1.B represents 25%, and thread A1.C represents 20%.

During this thread iteration the risks identified in the risk analysis report are monitored but their priorities are unchanged so no risk management techniques are necessary.

The design of the test cases begins during the later part of the design phase, when the design information is fairly stable. These test cases are both unit tests and functional tests. Implementation of the test cases begins during the implementation phase. After the initial implementation of the software components for thread A1, the test cases are exercised, during which time a number of defects are identified. The identified defects are resolved and the test cases are again executed. This process is repeated until all of the test cases are performed without generating any errors. It is important to note that the implementation phase is not completed until all defects are resolved.

Once all of the testing is completed, the requirements of the thread are traced to the resulting software, verifying that the requirements are met. At this point the basic architecture of the system is in place and a skeletal version of the overall system exists with the client application being able to connect to the server application, which is able to connect to the RDBMS, and basic messages are passed between the client and server applications.

Halfway through the first iteration, the project manager requests a progress report for the project. First, the manager of thread A1 asks the managers of threads A1.A, A1.B, and A1.C for the progress estimates. A1.A reports 40% complete, A1.B reports 60% complete, and A1.C reports 75% complete. The manager of thread A1 then computes the overall progress of the implementation phase of A1 to be 52% $((55\% * 0.4) + (25\% * 0.6))$

+ (20% * 0.75) = 22% + 15% + 15% = 52%). The requirements analysis, planning, analysis, and design phases have all been completed at this point, and the thread manager estimates that these comprise 50% of the overall effort of thread A1. The thread manager has also estimated that the implementation phase represents 30% of the overall effort of thread iteration A1 and the testing phase (which has not yet begun) represents 20% of the overall effort of thread iteration A1. Therefore, the overall progress of thread A1 is 65.6% ((50% * 1.0) + (30% * 0.52) + (20% * 0.0) = 50% + 15.6% + 0% = 65.6%).

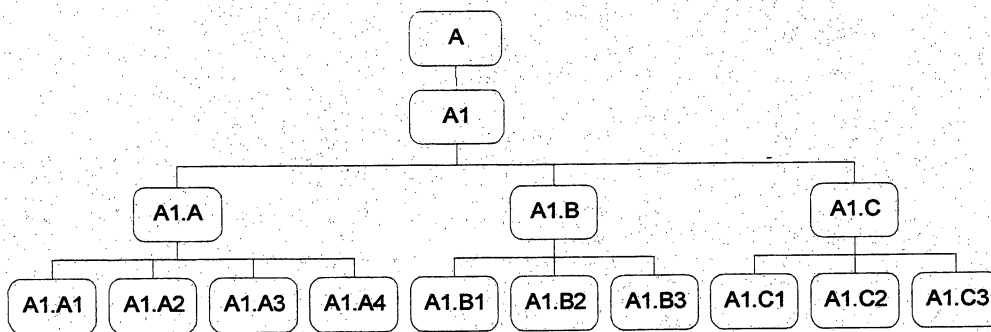


Figure 4.1: Thread Hierarchy of the First Iteration.

4.3 The Second Iteration

The first iteration of the root thread, A1, implemented the basic architecture of the system. The goal of the second iteration is to add a GUI to the client application and to implement the query operation. The unique name of the second iteration of the root thread is A2.

Because the first iteration of the root thread was an internal milestone and no discrepancies were identified with the system requirements, no modifications to the system

requirements are necessary. Because there were no changes to the requirements document from the previous iteration, the development plan does not need to be modified. It does need to be reviewed to determine how the actual progress of the system is relative to the development plan. At this time, development progress is on schedule according to the estimates in the development plan.

The analysis phase of this thread iteration involves specifying the problems for constructing the client application GUI to enable the client to perform queries against the RDBMS. These problems are independent of each other and are analyzed separately. The analysis of the client GUI involves the description of what user interface elements (e.g., windows, buttons, etc.) should exist. The analysis of the requirement to query the RDBMS involves substantially more objects that affect the client, server, and the database subsystems.

The design for the client GUI involves describing, in detail, what user interface elements should comprise the GUI, how they should look, how they should be organized, and how they should behave. The design for the client, server, and database subsystems involves identifying all of the classes required to implement the requirements for each module, and specifying their attributes, behavior, and interactions. Because there is already an existing system (produced during the first iteration, A1) that the new implementation will be added to, the design documentation and source code for the existing software is reviewed and the new design is added and incorporated into the design document from the previous iteration.

After the second iteration is completed, the system is delivered to the user for preliminary use and feedback. During this time the user identifies several minor aesthetic issues with the client GUI, but also identifies an additional requirement that was not included in the previous requirements document. After being able to query existing data in the RDBMS, the user realized that they would need to update existing data in addition to inserting new data. This requirement is added at the beginning of the next iteration.

4.4 The Third Iteration

The original goal of the third iteration was to add the ability to insert data into the RDBMS from the client application. However, after the delivery of the software built during the second iteration, the user identified a new requirement for the software to update data in the RDBMS. The change of requirements means that system requirements must be reviewed and re-evaluated before development can continue. The unique name of the third iteration of the root thread is A3.

After additional meetings with the user to fully define the new requirement, the requirements document is updated to reflect the changes identified by the user. These changes consist of the new requirement to update data in the RDBMS from the client application and several slight changes to the client application's GUI. Because of the change to the system requirements, specifically the addition of new requirements, the user is notified to expect an impact to the schedule and cost of the project.

As a result of changing the system requirements, the development plan must be modified to reflect these changes. This process is essentially the same as the first iteration with the exception that a number of requirements may have already been satisfied by

previous iterations (and can be ignored if they are not affected by the changes) and there may already be resource estimates for existing requirements. Each of the remaining requirements to be implemented must be re-prioritized, the scheduled number of root thread iterations must be updated according to the new set of requirements, and the remaining requirements to be implemented must be assigned to the remaining iterations.

Because the addition of the requirement to update data in the RDBMS does not affect the existing implemented system or the remaining requirement for inserting data into the RDBMS, a new thread iteration is added to implement the new requirement. The requirement for updating data in the RDBMS is deemed more significant to the user, so it is scheduled for implementation during the third iteration and the requirement to insert data into the RDBMS is scheduled for the fourth iteration.

The remainder of development during the third iteration proceeds similar to previous thread iterations, without any major difficulties. The resulting system is delivered to the user for evaluation and no new changes are identified.

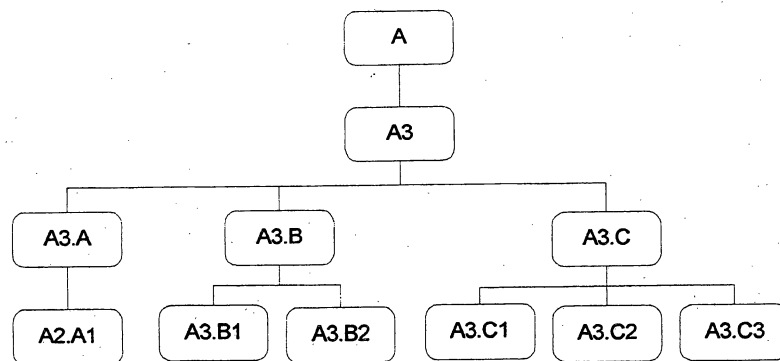


Figure 4.2: Thread Hierarchy of the Third Iteration.

4.5 The Fourth Iteration

The primary requirement of the fourth iteration is to implement the insertion of data into the RDMBS from the client application. The requirements for the fourth iteration are unchanged from the third iteration, so the requirements analysis and planning phases are uneventful. The analysis, design, implementation, and testing are performed without any incidents and the resulting system is delivered to the user. At this point the delivered system satisfies all of the system requirements, satisfying the developers contract with the user and the project is complete.

4.6 Additional Considerations

The previous example shows a typical application of RMT to a project. There are a number of situations that may arise during development that require additional considerations.

4.6.1 Iterations and Child Threads

Dividing the implementation of a particular software component into a number of iterations and creating multiple child threads is intended to make the software development process easier. While these techniques can be very helpful, they can also have negative effects if they are misused. Each thread iteration or new child thread requires some amount of overhead to manage. If many thread iterations are used to implement a small software component, then the effectiveness of using iterations is diminished because more effort is spent managing the iterations than is gained by using iterations. The same applies to child threads. In addition, the more levels that exist in the thread hierarchy of a project, the more potential there is for the loss of conceptual integrity of the system

because of the number of individuals involved in communicating information. Therefore, thread iterations and the spawning of child threads should be used only when the iteration and/or child threads results in more effort saved than is spent managing the iteration and/or thread (chapter three discusses the benefits of iterations and child threads).

4.6.2 Early Termination of a Thread (Handling Inconsistencies/Defects)

No matter how good a process or methodology is, or how skilled the developers are, people still make mistakes. The architect may overlook some obscure detail, designers may produce poor designs, and engineers may introduce defects during implementation. As a result of these mistakes, the development plan must be altered to resolve these problems. This may have a relatively small impact, affecting a single phase of development, or it may have significant repercussions, affecting the entire project.

The RMT life-cycle is designed to help reduce the impact of changes in the development plan by using incremental development, promoting open-ended architectures, etc. These techniques do not always accommodate all changes so seamlessly. There are situations that may arise after a thread has begun that causes the typical thread life-cycle to be altered to address these changes or defects. Some of these situations include:

- The identification of inconsistencies, flaws, or defects in the requirements, development plan, problem specification, design specification, or software implementation that affects an ancestor thread.
- A risk identified in the risk analysis report that either occurs or the probability/severity of becomes so great that it must be resolved immediately.

- A change of requirements occurs.

In all of the above situations, the current thread is immediately terminated and the parent thread is notified of the problem, and the issue must be resolved by some higher-level thread. If the problem identified is related only to the parent thread, sibling threads may or may not continue as planned, depending upon the nature of the problem. If the problem is not limited to the parent thread, then the parent thread (and all of its child threads) is also terminated and its parent thread is notified. The problem is then propagated up the thread hierarchy until it can be resolved by the appropriate thread. It is also possible that the identified problem may affect other threads not directly related to thread that identified the problem.

Once the situation has been resolved, the thread at which the problem was resolved begins a new thread iteration. This new thread iteration is similar to the initial iteration because it needs to re-evaluate the thread requirements because they may have changed as a result of the resolved problem. This may cause changes to the development plan as well as child threads. Child threads that were previously planned may be eliminated, new child threads may be required, and previously planned child threads may continue with different requirements.

4.6.3 Managing Multiple Abstraction Levels

With the potential for a large number of threads, sub-threads, etc., and associated development teams, it is important to have good communication between team members so that questions and problems can be addressed quickly and efficiently. This can be done by identifying well-known channels of communication between teams. Within RMT, each

thread has a manager who, among other responsibilities, is the primary contact for questions and issues related to that thread. If a team member has an issue with a particular thread they can raise the issue with that threads manager. In addition to answering questions and handling problems, the thread manager is responsible for reporting progress regularly to the thread manager of the parent thread. This allows the manager of the parent thread to update own their progress estimates. If problems are identified during a thread that can not be resolved by the current thread manager, the problem is discussed with the manager of the parent thread.

4.6.4 Methodologies

RMT is a development process, not a methodology. The two are orthogonal and the methodology (or methodologies), used during development can be chosen independently of the process. RMT does not require or enforce the use of any particular methodology during development. Information is communicated between RMT thread phases in the form of documentation, whose form and content is dictated by the particular methodology (or methodologies) used during each phase. The information input to certain thread phases may have particular content and format constraints based upon the requirements of a methodology. If different methodologies are used for two thread phases that exchange information, the information must be compatible between methodologies. If the information communicated between thread phases is not compatible, is lacking in detail, or contains too much detail, it must be modified to a format usable by the methodology used in the receiving thread phase. The conversion of information between

methodologies can require additional effort and resources, and has the potential for misinterpretation and loss of information.

While RMT does not advocate the use of one or more methodologies, because of the additional effort and potential for miscommunication of information during translation between methodologies, it is suggested that a single methodology be used throughout the life-cycle, if possible. If more than one methodology is used, great care should be taken in choosing methodologies that require little or no translation to provide an efficient transition between development activities.

The Unified Modeled Language (UML) has recently emerged as a language for specifying, visualizing, and constructing software that is based upon existing proven methodologies [Booch-97]. One of the benefits (and goals) of UML is that it provides a single “unified” perspective across development phases, eliminating the overhead of translating information between methodologies and notations. UML is largely based upon Jacobson’s Object-Oriented Software Engineering (OOSE) method [Jacobson-92], the Booch method [Booch- 94], and the Object Modeling Technique (OMT) [Rumbaugh-91]. Each of these methods has notably different strengths in different development activities: OOSE provides excellent requirement analysis capabilities, OMT is exceptionally expressive for analysis of information systems, and Booch-’93 is expressive during the design and construction of software. UML incorporates the best aspects of each of these methods and presents them in a seamless model. This makes UML an excellent candidate methodology for use within the RMT life-cycle.

Chapter Five-Conclusions

This thesis has shown that existing software development life-cycles do not support monitoring progress during the development process and they do not satisfy the requirements of developing object-oriented software (outlined in chapter one). Object-oriented life-cycles do not adequately support progress monitoring and traditional life-cycles do not accommodate the general needs of object-oriented development. Because of the need for a life-cycle to support these requirements, RMT was developed. RMT is a complete software life-cycle, borrowing several positive qualities from several existing life-cycles, which encompasses all the phases during the lifetime of a software system, from its conception to final delivery and maintenance. The most significant contribution of RMT is its ability to support progress monitoring through the use of threads as an abstraction to organize development activities. In addition to defining the components of RMT (chapter three), the application of RMT to a hypothetical project was presented (chapter four).

Even though RMT does address the needs of object-oriented development, it is not Brook's "silver bullet", having its own strengths and weaknesses. The biggest weakness of RMT is that it is a theoretical life-cycle that has not yet been proven through use on a real-world project. Even though it has not been exercised in a real-world situation, the core concepts of RMT are similar to other "proven" life-cycles, so it is anticipated that the results would be successful. Another weakness of RMT is the potential for an exponential explosion of threads and thread iterations by misusing recursion and iteration. To help guard against this problem, guidelines should be established by an organization to help

prevent this from happening and to identify, at an early stage, when a problem does occur so that it can be corrected before the problem becomes unmanageable.

Even though a project is object-oriented, the stability of the system requirements can influence the benefits of RMT. David Bond [Bond-95] has presented four major categories of software development projects based upon the source of the requirements and the number of clients. In order of most stable to least stable requirements, they are: constrained software, internal client software, vertical market software, and mass market software. Constrained software has highly constrained requirements at the beginning of the project that remain unchanged during development and is generally built for one customer. At the opposite end of the spectrum, mass market software is built for a large number of customers, has frequently changing requirements, and has high scheduling pressures dictating the functionality that is included at the time of release.

While thread iterations and recursion can be applied to any project, RMT (and iterative life-cycles in general) is most appropriate for projects where the system requirements are vague or frequently changing, like Bond's mass market software classification. RMT can still be used effectively for the other project types, but the iteration and recursion techniques can be used as internal development styles rather than a means to accommodate changing requirements and/or schedules.

In addition to the stability of system requirements, RMT is most useful for medium- to large-scale projects rather than small-scale projects. This is because for small-scale projects the benefits of using RMT are outweighed by the overhead required to manage the threads.

5.1 Relevance to the Capability Maturity Model (CMM)

The current version of the CMM, v1.1 [Paulk-93a, Paulk-93b], was developed by the Software Engineering Institute (SEI) at Carnegie-Mellon University which defines a model for process maturity used by an organization. The CMM defines an evolutionary path for process maturity, so that an organization can more easily improve its development process. Each step, or level, in the evolutionary path is built upon previous steps, providing additional improvements, and requires the presence of certain key activities, techniques, and tools called key process areas (KPAs). The five levels of maturity, in increasing order of maturity, are: initial, repeatable, defined, managed, and optimizing.

Table 5.1 summarizes each maturity level.

| Maturity Level | Name | Description |
|----------------|------------|--|
| 1 | Initial | The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort. |
| 2 | Repeatable | Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications. |
| 3 | Defined | The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and maintaining software. This level indicates all |

| | | |
|---|------------|--|
| | | characteristics defined for level 3. |
| 4 | Managed | Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3. |
| 5 | Optimizing | Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4. |

Table 5.1: CMM Maturity Levels [Pressman-97]

Some of the KPAs required for various CMM maturity levels are concerned with organizational and management techniques for the software development process such as software project planning, requirements management, etc. Software life-cycles, like RMT, address many of these same KPAs. Other CMM KPAs are targeted towards the overall development approach of an organization that are outside the scope of a software life-cycle, such as peer reviews, training programs, and technology change management. Because RMT only addresses a subset of the KPAs required for all five levels of maturity, RMT can not solely satisfy the requirements for all five levels of CMM maturity. RMT supports most (but not all) KPAs of maturity levels two and three, but none of levels four and five. The RMT process, however, does not exclude a developer from any of the maturity levels (i.e., using RMT does not prevent a developer from qualifying for a particular maturity level).

Simply using RMT does not imply that an organization will automatically be compliant with a particular CMM maturity level. When used in conjunction with several additional software engineering practices, RMT provides a strong foundation for being compliant with the CMM. For levels three and four, RMT provides the foundation for a majority of the CMM requirements.

5.2 Future Directions

There are a number of future directions and tasks that research for RMT can (and should) take. The most important step in the evolution of RMT is its application to a real-world project. A project should be selected that is a medium- to large-scale project with loosely-defined or changing requirements. It would also be of particular interest to somehow measure the effectiveness of RMT, possibly comparing it with the effectiveness of other life-cycles. The successful application of RMT would give it more credibility, moving it out of the domain of theoretical life-cycles to a practical life-cycle.

Another area of interest would be to develop a computer aided software engineering (CASE) tool (using the RMT process itself to develop the tool) to model and document the RMT development process. This would allow project managers and developers to easily review and update any aspect of the development process (e.g., update resource estimates, revise delivery dates, etc.). The example in chapter four hints at some requirements for such a CASE tool: being able to graphically display thread iterations in a project hierarchy, display/edit property information for a thread iteration, allow multiple users access to the same project information, etc.

It would also be useful to describe how to apply UML diagrams and notations during each of the RMT phases. This would provide developers with a practical step-by-step “cookbook” on how to apply RMT and UML to their own project.

Finally, there is interest in developing a system for maintaining a repository of design pattern information (at California State University, San Bernardino) that could be done in conjunction with the development of an RMT CASE tool. Because RMT suggests the use of design patterns, perhaps an RMT CASE tool could directly interface with such a design pattern repository system. The requirements of the CASE tool could influence the requirements and design of such a system.

Appendix A-Glossary

Because many terms are used by different individuals with different meanings, this appendix provides definitions for terms used throughout this thesis to avoid any ambiguities in their interpretation.

abstraction - A view of an object, entity, or other conceptual element that only considers the characteristics relevant or necessary for a particular purpose while ignoring the remaining, irrelevant characteristics.

activity - An operation or technique that is performed to complete some goal during a particular phase in a life-cycle (see also, task).

bottom-up design - The process of designing a system by starting with the most primitive abstractions or components and progressively building higher-level abstractions to the highest-level component (contrast with top-down design).

class - An abstraction that represents the logical collection of entities or objects with similar attributes and behaviors.

cohesion - The degree which functions, procedures, or operations within a given module are “functionally” related.

component - A collection of one or more classes, a module, or a subsystem.

coupling - The degree which modules are related to or dependent on other modules.

divide-and-conquer - A problem-solving technique which “divides” a problem into a number of smaller pieces, recursively applies the technique to each piece, then combines the results into a single solution.

encapsulation - The process of grouping both the structure and behavior of an abstraction, usually to separate the interface of the abstraction from its implementation.

evolutionary development - The incremental development of a software system where each increment produces a version of the software that extends, enhances, or improves previous versions of the software. This is similar to the “evolution” of biological organisms over time.

iteration - The process of repeating a series of development phases during the development of a software component to extend, enhance, or improve the implementation of the component.

life-cycle - (a.k.a. software life-cycle, development life-cycle, development process) A systematic process that can be applied during the construction of software. A life-cycle usually divides construction into a number of phases which have very well-defined goals, tasks, inputs, and outputs (e.g., analysis, design, implementation).

methodology - A particular approach or technique that can be used to solve a particular class of problems, such as analysis or design. Methodologies are generally used within a life-cycle phase.

model - An abstraction that is used to clarify or understand a complex artifact, such as software systems or real-world scenarios.

module - A program unit which is some logical collection of operations or objects.

modularity - The property of discrete components that are highly cohesive and loosely coupled.

object - A particular instance of a class which contains its own unique attribute values.

phase - A period of time within a life-cycle, during which a number of predefined

activities or tasks are performed to carry out some well-defined goal.

process - The definition and organization of the activities performed during the

development of a software system.

requirement - A capability, condition, or functionality that is needed to achieve some

identified goal. System requirements specify the functionality required by a

software system to satisfy the needs of the user.

software development - The process of conceiving and implementing a software system.

structured design - The process of designing by algorithmic decomposition.

task - An operation or technique that is performed to complete some goal during a

particular phase in a life-cycle (see also, activity).

thread - An abstraction which represents the development of a software component to

satisfy some set of requirements. It distinguishes several activities, or phases, that

have well-defined goals, preconditions, and postconditions during the actual

component development. A thread may be iterated any number of times to

incrementally implement the required software component(s). In addition, a thread

iteration may create a number of other threads to implement lower-level

components. The same step-by-step process defined by a thread is applied to many

different parts of a project by many different developers with different skills and

responsibilities.

top-down design- The process of designing a system by starting with the highest-level component and proceeding to lower-level components through a hierarchy.
(contrast with bottom-up design).

Bibliography

- [Alfred-95] Charlie Alfred and Stephen J. Mellor, Observations on the role of Patterns in Object-Oriented Software Development, *Object Magazine*, 5(2), May 1995, pp. 61-65.
- [Berard-93] Edward V. Berard, *Essays on Object-Oriented Software Engineering Volume 1*, Prentice Hall, 1993.
- [Boehm-88] Barry W. Boehm, A Spiral Model of Software Development and Enhancement, *IEEE Computer*, May 1988, pp. 61-72.
- [Bond-95] David Bond, Project-Level Design Archetypes, *Software Development*, July 1995, Vol. 3, No. 7.
- [Booch-91] Grady Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, 1991.
- [Booch-94] Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin/Cummings, 1994.
- [Booch-97] Grady Booch, et al., *Unified Modeling Language: UML Summary*, v1.0, Rational Software Corporation, January 1997, <http://www.rational.com>.
- [Botting-97] Richard J. Botting, Personal Communication, 1997.
- [Brooks-95] Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition, Addison Wesley, 1995.
- [Coplien-94] James O. Coplien. *Software Design Patterns: Common Questions & Answers*, Proceedings of Object Expo New York, pages 39-42, June 1994, New York, SIGS Publications.
- [Cusumano-95] Michael A. Cusumano and Rick Selby, *Microsoft Secrets*, Simon & Schuster, 1995.
- [Fang-96] F.W. Fang, Andrew C. So, R. Jordan Kreindler, The visual modeling technique: An introduction and overview, *Journal of Object-Oriented Programming*, July/Aug 1996, Vol. 9, No. 4.
- [Gamma-95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [Gilb-88] T. Gilb, *Principles of Software Engineering*, Addison-Wesley, 1988.

- [Henderson-Sellers-90] Brian Henderson-Sellers and Julian M. Edwards, The Object-Oriented Systems Life Cycle, Communications of the ACM, September 1990 Vol. 33, No. 9.
- [Henderson-Sellers-94] Brian Henderson-Sellers and J.M. Edwards, Book Two of Object-Oriented Knowledge: The Working Object, Prentice Hall, 1994.
- [Isakowitz-96] Tomás Isakowitz and Robert J. Kauffman, Supporting Search for Reusable Software Objects, IEEE Transactions on Software Engineering, June 1996, Vol. 22, No. 6.
- [Jacobson-92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard, Object-Oriented Software Engineering A Use Case Driven Approach, Addison-Wesley, 1992.
- [Leveson-93] N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," IEEE Computer 26 (July 1993), pp. 18-41.
- [Lientz-78] B.P. Lientz, E.B. Swanson, and G.E. Tompkins, Characteristics of Application Software Maintenance, Communications of the ACM 21, June 1978, pp.466-471.
- [Lewis-96] Bil Lewis and Daniel J. Berg, Threads Primer: A Guide to Multithreaded Programming, Prentice Hall, 1996.
- [McConnell-96] Steve McConnell, Daily Build and Smoke Test, IEEE Software, July, 1996, pp. 144, 143.
- [McGregor-92] John D. McGregor, David A. Sykes, Object-Oriented Software Development: Engineering Software for Reuse, Van Nostrand Reinhold, 1992.
- [Mellor-94] P. Mellor, "CAD: Computer-Aided Disaster," Technical Report, Centre for Software Reliability, City University, London, UK, July 1994.
- [Meyer-88] Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, 1988.
- [Meyer-89] Bertrand Meyer, .From structured programming to object-oriented design: the road to Eiffel, Structured Programming, 1, 1989, pp. 19-39.
- [Mili-95] Hafeedh Mili, Fatma Mili, and Ali Mili, Reusing Software: Issues and Research Directions, IEEE Transactions on Software Engineering, Vol. 21, No. 6, June 1995, pg 528-562.
- [Moore-96] James W. Moore and Roy Rada, "Organizational Badge Collecting," Communications of the ACM, August 1996, vol. 39, no. 8, pp. 17-21.

- [Naur-69] Naur, P., and B. Randell (eds.), *Software Engineering: A Report on a Conference sponsored by the NATO Science Committee*, NATO, 1969.
- [Paulk-93a] M.C. Paulk, B. Curtis, M.B. Chrissis, and C.V. Weber, *Capability Maturity Model for Software, Version 1.1*, Software Engineering Institute, CMU/SEI-93-TR-24, February 1993.
- [Paulk-93b] M.C. Paulk, C.V. Weber, S. Garcia, M.B. Chrissis, and M. Bush, *Key Practices of the Capability Maturity Model, Version 1.1*, Software Engineering Institute, CMU/SEI-93-TR-25, February 1993.
- [Pressman-97] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 4th ed., McGraw-Hill, 1997.
- [Raccoon-95] L. B. S. Raccoon, *The Chaos Model and the Chaos Life Cycle*, *Software Engineering Notes*, January 1995, Vol. 20, No. 1.
- [Royce-70] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," 1970 WESCON Technical Papers, Western Electronic Show and Convention, Los Angeles, August 1970, pp. A/1-1-A/1-9.
- [Rumbaugh-91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [Saiedian-95] Hossein Saiedian and Richard Kuzara, "SEI Capability Maturity Model's Impact on Contractors," *IEEE Computer*, January 1995, pp. 16-26.
- [Schach-96] Stephen R. Schach, *Classical and Object-Oriented Software Engineering*, 3rd ed., Irwin, 1996.
- [Shaw-84] M. Shaw, *Abstraction Techniques in Modern Programming Languages*, *IEEE Software*, vol. 1(4), October 1984, p. 10.
- [Singh-95] Raghu Singh, "The Software Life Cycle Processes Standard," *IEEE Computer*, November 1995, vol. 28, no. 11.
- [Swanson-76] Swanson E. B., *The Dimensions of Maintenance*, *Proc. 2nd Intl. Conf. On Software Engineering*, IEEE, October 1976, pp. 492-497.
- [Wilkinson-95] N. Wilkinson, *Using CRC Cards: An Informal Approach To Object-Oriented Development*, SIGS Books, New York, 1995.
- [Wirfs-Brock-90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

[Yourdon-92] E. Yourdon, *Decline and Fall of the American Programmer*, Yourdon Press, Englewood Cliffs, 1992.

[Zelkowitz-79] M.V. Zelkowitz, A.C. Shaw, and J.D. Gannon, *Principles of Software Engineering and Design*, Prentice-Hall, Englewood Cliffs, 1979.