SOFTWARE ENGINEERING METHODOLOGY: THE WATERSLUICE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Ron Burback
December 1998

© Copyright 1999 by Ron Burback

All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Gio Wiederhold
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David Luckham

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Eric Roberts

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dorothea Beringer

Approved for the University Committee of Graduate Studies:

Preface

The body of methods, rules, postulates, procedures, and processes that are used to manage a software engineering project are collectively referred to as a methodology.

There are two well-known software engineering methodologies commonly used in practice today. The two methodologies, informally known as the waterfall and spiral methodologies, are characterized by the grouping of tasks as either sequential or cyclical. Both of these methodologies organize some tasks very well, but have a narrow focus, so that crucial aspects of one methodology are missed by the other methodology.

This thesis defines the WaterSluice methodology. The WaterSluice borrows the iterative nature of the cyclical methodology along with the steady progression of the sequential methodology. In addition, the tasks in the WaterSluice are prioritized such that the most beneficial, non-conflicting tasks are accomplished first. A collection of theorems is presented establishing the strengths and weaknesses of the WaterSluice as compared to the sequential and cyclical methodologies. The WaterSluice methodology represents a more accurate rendering of current software engineering practices. In this sense, the WaterSluice is not new but merely represents a concise description of the state of the art.

This thesis builds a foundation for the study of software engineering methodologies and then categorizes the conditions under which one software engineering methodology will be preferred over another software engineering methodology. Predicted performance characteristics for several major classes of software engineering methodologies under a variety of conditions are presented.

Software engineering is a large and complex process of many interdependent processes of which the methodology is only one part. This thesis concentrates on the methodologies, but does not ignore many of the other key issues. In the appendices, other key issues are covered including issues associated with requirement gathering including an example of a requirement document, the software engineering system life-cycle, the software engineering five-level engineering support environment, decision support, and documentation development.

After the introductory Chapter 1, Chapter 2 introduces the foundation phases of analysis, design, implementation, and testing. Chapter 3 builds from the foundation phases, the sequential, cyclical, and WaterSluice software engineering methodologies. Several established methodologies are reviewed in Chapter 4. Chapter 5, the formal foundation chapter of the thesis, establishes the theoretical results which are then compared to similar results from search theory in Chapter 6. This is followed by a survey of large software engineering projects in Chapter 7. The final chapter of the thesis, Chapter 8, introduces future work in distributed architectures, environments, paradigms, and component engineering. The other topics as mentioned above are then covered in the appendices.

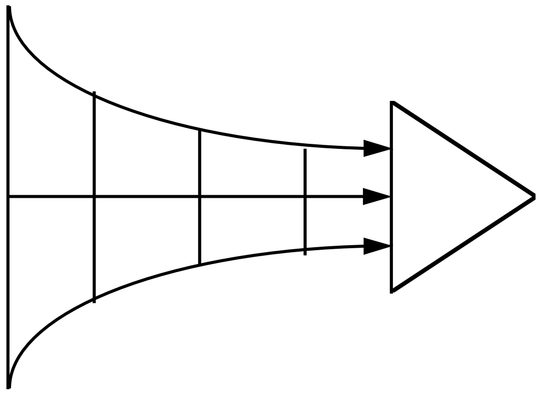


Figure 0.1: The WaterSluice Icon

Acknowledgments

I would like to acknowledge my principal thesis advisor, Gio Wiederhold, because without his help this thesis would not have been possible. I would also like to thank my reading committee members: David Luckham, Eric Roberts, and Dorothea Beringer. A special thanks goes to Catherine Tornabene, Gary Payne, and Jie Wang for their helpful suggestions.

I would also like to thank my wife, Sandy, for the loving and understanding support given to me while I pursued my educational studies. I would also like to acknowledge my powerful daughters, Jennifer, Christy, and Katy, who almost graduated from college before their father.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A **software engineering** project involves people guided by common goals and strategies working with a collection of tools to produce documents and code. The tools include compilers, debuggers, environments, change management, source control, project management, document processors, and domain modeling tools. The documents produced include requirements that define the problem, customer manuals, test plans, scenarios, a design that defines the architecture, and implementation plans. The code may deal with objects, data structures, algorithms, methods, modules, protocols, and interface definitions. The strategies are materialized through the collection of the architecture, methods, paradigms, risk analyses, conventions, and a mission statement. These steps together define the cradle-to-grave life cycle of the software project.

Just how should a software engineering project be managed? The answer is not unique nor is it clearly defined. It is a combination of many ingredients. One of the key ingredients in the management of an engineering project is the **methodology**.

Two well-known categories of software engineering methodologies are the **sequential** and the **cyclical**, informally known as the waterfall and spiral methodologies. A third methodology is introduced here, called the **WaterSluice** methodology, which combines the best aspects of the previous two methodologies. It takes the iterative nature of the cyclical methodologies and the steady progression of the sequential methodologies and then adds priority and conflicting requirement management.

All three methodologies deal with four simple **phases** of software engineering, namely **analysis**, **design**, **implementation**, and **testing**. These concepts are first introduced before the detailed discussion of methodologies.

The following is the main theorem of the thesis.

Theorem 1 *Different software engineering methodologies have significant performance variations depending on the given environment. A software engineering methodology that is goal focused, manages conflicts, and differentiates between different priorities is best suited for dynamic non-monotonic environments.*

Though the statement of the theorem appears simple, the complexity is in the details of the formal definitions of software engineering methodologies, performance, and environment. The variations in performance of different software engineering methodologies are sufficiently great as to make the choice of which software engineering methodology to use dependent on the surrounding environment.

First the **environment** will be defined. As discussed later, the environment definition is built from the definitions of the analysis, design, implementation, and testing phases. Each phase defines a **plane** which is then defined in terms of **atomic**, **compound**, and **complex steps**. One step may have a **sibling relationship** with other steps. Together the four planes form a **multi-layered space**, either **static** or **dynamic**. In some cases, a dynamic space may exhibit the **non-monotonic** property.

There are two special steps: the **problem statement** and the **system acceptance test**. The environment is the multi-layered space consisting of the analysis, design, implementation, and testing planes with two special steps: the initial problem statement and the system acceptance test.

Next we define the methodology, as discussed later, as an algorithm that finds a **solution** in the given environment of the multi-layered space consisting of the analysis, design, implementation, and testing plane starting with the root represented by the problem statement and ending with the goal represented by the system acceptance test. Three classes of methodologies are presented: sequential, cyclical, and the WaterSluice.

Finally, we define performance, as discussed later, as the number of steps needed by a methodology to find a solution. A family of theorems and corollaries are then proven. In summary, the proceeding theorems and corollaries generate several key results:

- All three categories of software engineering methodologies are complete for static environments. See Theorems 2, 3, and 4.
- Only cyclical and WaterSluice are complete for dynamic environments. See Corollaries 2.4, 3.4, and 4.1.
- Only WaterSluice is complete for non-monotonic environments. See Corollaries 2.5, 3.5, and 4.2.
- The best case performance of sequential software engineering methodology is O(N). See Corollaries 2.1.
- The best case performance of cyclical and WaterSluice software engineering methodologies is O(1). See Corollaries 3.1, and 4.3.

- The worst case performance of all three categories of methodologies are the same. See Corollaries 2.2, 3.2, and 4.4.
- On average, the sequential methodology will find a solution in O(N). See Corollary 2.3. On average, the cyclical methodology will find a solution in O(N). [1] See Corollary 3.3. On average, the WaterSluice will find a solution an order-of-magnitude less than N. See Corollary 4.5.
- A software engineering methodology that is goal focused, manages conflicts, and differentiates between different priorities is best suited for dynamic non-monotonic environments.

As an analogy, the search section compares the three methodologies to algorithms in a search space. It will be shown that the sequential methodologies is analogous to a **breadth-first search**, the cyclical methodologies is analogous to a **depth-first search**, and the WaterSluice methodology is analogous to a **best-first search**.

Some established methodologies are reviewed in Chapter 4 including Boehm-Waterfall Methodology, Boehm-Spiral Methodology, Booch Methodology, Object Modeling Technique (OMT), Rational Objectory Methodology, and WinWin Spiral Methodology.

Chapter 7 reviews several large projects while future work is outlined in Chapter 8.

[1] A more accurate average performance measurement for a cyclical software engineering methodology is N/2.

Chapter 2

Software Engineering Phases

2.1 Introduction

Before we descend into the definition of software engineering methodologies, we need to define the meanings of some of the fundamental phases.

There are four fundamental **phases** in most, if not all, software engineering methodologies. These phases are analysis, design, implementation, and testing. These phases address what is to be built, how it will be built, building it, and making it high quality. These phases will now be defined as they apply to the life cycle stage of product delivery emphasized in this thesis.

Even though this thesis emphasizes the four phases of analysis, design, implementation, and testing in a software engineering methodology as it applies to the software **life cycle stage** of **product delivery**, the results are also applicable to the other software life cycle stages of **deployment**, **operations**, **maintenance**, **legacy**, and finally **discontinuation** as the system transitions through many versions from cradle to death. This is explored in more details in Appendix A. In many systems the majority of the total system cost is in the later life cycle stages and only a minority of the total system cost in the initial development.

2.2 The Analysis Phase

Phase	Deliverable
Analysis	• Requirements Document • Domain Ontology - Things - Actions - States • Typical Scenarios • Atypical Scenarios

Table 2.1: The Analysis Phase: What does the system do?

The **analysis phase** defines the **requirements** of the system, independent of how these requirements will be accomplished. This phase defines the problem that the customer is trying to solve. The deliverable result at the end of this phase is a requirement document. Ideally, this document states in a clear and precise fashion what is to be built. This analysis represents the "what" phase. The requirement document tries to capture the requirements from the customer's perspective by defining goals and interactions at a level removed from the implementation details. The analysis phase is summarized in Table 2.1 on page 6.

The **requirement** document may be expressed in a formal language based on mathematical logic. Traditionally, the requirement document is written in English or another written language.

The requirement document does not specify the architectural or implementation details, but specifies information at the higher level of description. The problem statement, the customer's expectations, and the criteria for success are examples of high-level descriptions. There is a fuzzy line between high-level descriptions and low-level details.

Sometimes, if an exact engineering detail needs to be specified, this detail will also appear in the requirement document. This is the exception and should not be the rule. These exceptions occur for many reasons including maintaining the consistency with other established systems, availability of particular options, customer's demands, and to establish, at the requirement level, a particular architecture vision. An example of a low-level detail that might appear in the requirement document is the usage of a particular vendor's product line, or the usage of some accepted computer industry standard, or a constraint on the image size of the application.

There is a fundamental conflict between high levels and low levels of detail. The requirement document states what the system should accomplish, independent of many of the details. The discovery process used in establishing the requirements during the analysis phase is best described as a refinement process than as a levels-of-detail process [106].

Top-down and bottom-up approaches force a greater distinction between high levels and low levels of detail. Interactive approaches lead to the refinement of those details.

Traditionally, the requirement document describes the **things** in the system and the **actions** that can be done on these things. Things might be expressed as objects in an object-based technology where data and algorithms are hidden behind hierarchical-polymorphic methods. [1] Alternatively, things might be expressed as services accessing

[1] In object-based systems a scheme forms an hierarchy used to establish inheritance of methods and data structures. A child object class in the hierarchy inherits their patent's methods and data structures. Multiple polymorphic methods share the same name and similar, conceptual algorithms.

databases in a functional approach where data is a fundamentally different concept than functions. In general, the description of things in the system can be much more general and not confined to a particular technology. In a more general sense, this document describes the **ontology**, that is the noun phrases and the verb phrases, that will become the guidelines for defining the application specific protocol.

The requirement descriptions of the things in the system and their actions does not imply an architecture design rather a description of the artifacts of the system and how they behave, from the customer's perspective. Later, in the design phase, these requirement descriptions are mapped into computer science based primitives, such as lists, stacks, trees, graphs, algorithms, and data structures.

The description of the abstraction of the noun phrases and the verb phrases are not bound to the use of a written human language. Most written human languages are too vague to capture the precision necessary to build a system. Alternative descriptive mechanisms based on mathematical logic are sometimes more suitable but much more difficult to accomplish. Mathematical logic provides a scientific foundation for precisely expressing information. However, frequently in the real world, a precise description is not attainable.

Again the requirement document should state in a clear and precise fashion what is to be built. The definitive mechanism to author such a document, either formally or informally, has yet to be developed, although reasonable success has been achieved with existing methods including CASE tools and tools based on mathematical logic. See [41], [27], and [110].

Later, in the design phase, the very important decomposition of the problem leads

---

The method "plus" used for integer, real, and complex would be an example of a polymorphic method.

to the development of data structures and algorithms. A functional decomposition for a distributed environment leads to a natural split of the data structures and algorithms. Examples include distributed client-server systems, where a database holds the data in a server while the algorithms manipulating the data reside on the client. An object-based decomposition leads to a natural joining of data structures and algorithms forming objects with methods. The requirement documents should be independent of the decomposition technique.

The analysis team develops the requirement document, which talks about things and actions on things. This document should also include states, events, typical scenarios of usage, and atypical scenarios of usage. The definitions of things, actions, states, typical scenarios, and atypical scenarios follow this section. More detailed examples of a requirement document can be found later in this thesis. See Appendix E on page 206 for an example requirement document.

2.2.1 Things

The requirement document first of all defines the ontology of the system which is, in the more general sense, the noun phrases. Here the pieces and parts, constants, names, and their relationships to each other are specified.

2.2.2 Actions

The requirement document defines the **actions** that the system should perform. This is expressed, in the more general sense, as verb phrases. Methods, functions, and procedures are all examples of actions.

2.2.3 States

States are defined as a sequence of settings and values which distinguishes one time-space slice of a system from another slice. Every state-full system goes through a series of state changes. Example states include the initial state, the final state, and potentially many error states. Most of the states are domain specific.

States are associated with things in the system. An event triggers a potential state transition which may then lead to an action taken by the system.

2.2.4 Typical Scenarios

A **scenario** is a sequence of steps taken to accomplish a given goal. When the system is completed and the application is available, the customer should be able, in an easy and clearly specified manner, to accomplish all typical usage scenarios for the application.

The **typical scenarios** should represent the vast majority of uses for the system. The exact coverage of the system by the typical scenarios vary, but a 90 percent coverage is desirable. Obviously, a system with only one possible usage scenario will be easy to cover while a system with thousands of possible usage scenarios will be much harder to cover.

Frequently the 80/20 rule is invoked. Eighty percent of the functionality of a typical system is accomplished by twenty percent of the work. To accomplish the remaining minority functionality requires the vast majority of the work.

2.2.5 Atypical Scenarios

An **atypical scenario** is something that needs to be accomplished within the system, but only seldom. The actions have to be done correctly, but perhaps at lower efficiency. The customer should hope that an unexpected error condition is an atypical event. Nonetheless, the system should be able to deal with many categories of faults by using several established techniques, such as exception handlers, replications, process monitoring, and roll over. Atypical scenarios and typical scenarios share similar coverage.

2.2.6 Incomplete and Non-Monotonic Requirements

An entire enumeration of all of the requirements is not possible for nearly all real-life situations. Godel's incompleteness theorem of arithmetic says that there is no finite list of axioms that completely describe integer arithmetic. Expressed in our terminology, there is no finite list of requirements that would completely describe arithmetic. Since integer arithmetic is an underlying foundation of most computer hardware systems and software applications, and since we can't even enumerate the requirements for integer arithmetic, the task of completely enumerating a more complex system is certainly intractable.

In traditional logic, a theory is defined by a finite set of axioms. Theorems within the theory are valid sentences. If new axioms are added to the theory, the already existing theorems remain valid and the theory is extended into a new theory with new theorems added to the established theorems.

In **non-monotonic** logic, adding new axioms to the theory may invalidate existing theorems that were already proven. A new theory is created which is not a simple extension of the old theory, but a collection of new theorems and some of the established theorems.

The requirement gathering process is iterative in nature and more like non-monotonic logic than monotonic logic. An initial collection of requirements, the axioms of the system, define the capabilities, the theorems of the system. New requirements may lead to a collection of capabilities different than the established capabilities. New requirements may negate old solutions.

Early in the process, some requirements are established. As the process continues, other requirements are discovered which may be in conflict with earlier known requirements, thus leading a different system. See [95].

Unfortunately, as a system increases in size and complexity, the requirement gathering process becomes more and more intractable. This is especially true when the requirement gathering process is distributed across many individuals from many different disciplines.

2.3 The Design Phase

Phase	Deliverable
Design	• Architecture Document • Implementation Plan • Critical Priority Analysis • Performance Analysis • Test Plan

Table 2.2: The Design Phase: What are the plans?

In the **design phase** the **architecture** is established. This phase starts with the requirement document delivered by the requirement phase and maps the requirements into an architecture. The **architecture** defines the components, their interfaces and behaviors. The deliverable design document is the architecture. The design document describes a plan to implement the requirements. This phase represents the "how" phase. Details on computer programming languages and environments, machines, packages, application architecture, distributed architecture layering, memory size, platform, algorithms, data structures, global type definitions, interfaces, and many other engineering details are established. The design may include the usage of existing components. The design phase is summarized in Table 2.2 on page 12.

The architectural team can now expand upon the information established in the requirement document. Using the typical and atypical scenarios provided from the requirement document, performance trade-offs can be accomplished as well as complexity of implementation trade-offs.

Obviously, if an action is done many times, it needs to be done correctly and efficiently. A seldom used action needs to be implemented correctly, but it is not obvious what level of performance is required. The requirement document must guide this decision process. An example of a seldom used action which must be done with high performance is the emergency shutdown of a nuclear reactor.

Analyzing the trade-offs of necessary complexity allows for many things to remain simple which, in turn, will eventually lead to a higher quality product. The architecture team also converts the typical scenarios into a test plan.

In our approach, the team, given a complete requirement document, must also indicate **critical priorities** for the implementation team. A critical implementation priority leads to a task that has to be done right. If it fails, the product fails. If it succeeds, the product might succeed. At the very least, the confidence level of the team producing a successful product will increase. This will keep the implementation team focused. Exactly how this information is conveyed is a skill based on experience more than a science based on fundamental foundations.

The importance of priority setting will become evident in the theory chapter presented later.

2.3.1 Architecture

The architecture defines the components, interfaces, and behaviors of the system.

The components are the building blocks for the system. These components may be built from scratch or re-used from an existing component library. The components refine and capture the meaning of details from the requirement document.

The components are composed with other components using their interfaces. An interface forms a common boundary of two components. The interface is the architectural surface where independent components meet and communicate with each other. Over the interface, components interact and affect each other.

The interface defines a behavior where one component responds to the stimuli of another component's actions.

2.3.2 Implementation Plan

The **implementation plan** establishes the schedule and needed resources. It defines implementation details including programming languages, platforms, programming environments, debuggers, and many more.

The implementation plan could be considered as part of the design, which is the position taken here, or it could be considered as the first accomplishment in the implementation phase. One of the goals of the design phase is to establish a plan to complete the system. Thus it is very natural to include the implementation plan. Also, the trade-offs between alternative architectures can be influenced by differences in their implementation plans.

2.3.3 Critical Priority Analysis

The **critical priority analysis** generates a list of **critical tasks**. It is absolutely necessary to successfully accomplish a critical task. The project will succeed or fail based on the outcome of these tasks. Some projects may have more than one critical task.

There are two major categories of critical tasks. One category of tasks are associated with the building of the system. These are the critical tasks that the teams must accomplish well. An example might be a high-quality implementation of a critical section of code in the system.

The other category of critical tasks are associated with the system itself. These are the critical tasks that the system, once built, must accomplish well. An example might be the successful flying of an airplane under automatic pilot.

It is absolutely necessary to successfully accomplish both categories of critical tasks.

Not all methodologies have critical priority analysis as a well defined task. Later in the thesis it will be shown that the setting of priorities will play a significant role in methodology's performance characteristics. Critical priority analysis is one of the key features of the WaterSluice software engineering methodology.

2.3.4 Performance Analysis

Once given the typical scenarios from the requirement document, the system can be designed to meet performance objectives. Different system architectures will yield different predicted performance characteristics for each typical scenario. Depending on the usage frequency of the scenarios in the system, each architecture will have benefits and drawbacks with advantages and disadvantages. The trade-offs are then weighted to establish the system architecture. Frequently a system is designed to give fast response to an action initiated by a human customer at the expense of having to do more complex systems work such as including indexes, cache management, and predictive pre-calculations.

2.3.5 Test Plan

The **test plan** defines the testing necessary to establish quality for the system. If the system passes all tests in the test plan, then it is declared to be complete. If the system does pass all test then it is considered to be of high quality. The more complete the coverage of the system, the higher is the confidence in the system: hence the system's quality rises.

The test plan could be considered as part of the design, which is the position taken here, or it could be considered as the first accomplishment in the testing phase. One of the goals of the design phase, is to establish a plan to complete the system, thus it is very natural to include the test plan. Also the trade-offs between alternative architectures can be influenced by differences in their test plans.

One single test will exercise only a portion of the system. The coverage of the test is the percentage of the system exercised by the test. The coverage of a suite of tests is the union of the coverage of each individual test in the suite.

Ideally, 100 percent test coverage of the entire system would be nice, but this is seldom achieved. Creating a test suite that covers 90 percent of the entire system is usually simple. Getting the last 10 percent requires significant amount of development time.

For an example, consider the Basic Input/Output System (BIOS) built by IBM in the early 1980s as the foundation of the Disk Operating System (DOS) built by Microsoft. For performance reasons, the BIOS needed to be placed in a Read Only Memory (ROM) chip. Because the BIOS would be placed on a ROM, error patches would be nearly impossible. Thus 100% test coverage of the BIOS was dictated. The BIOS code itself was small, only a few thousand lines. Because the BIOS is asynchronous in nature, creating a test would first require an asynchronous environment to bring the system to a desired state, and then an event would be needed to trigger a single test. Quickly, the test suite grew much larger than the BIOS. This introduced the problem of doing quality assurance on the test suite itself. Eventually, 100% coverage was reached but at a high cost. A more cost effective approach would be to place the BIOS on a Electronic Programmable ROM (EPROM) and ROM combination. Most of the BIOS would be on the ROM with error patches being placed on the EPROM. This is the approach that Apple took on the Macintosh.

Usually, it is sufficient that the test suite includes all of the typical and atypical scenarios and need not cover the entire system. This gives reasonable quality for the investment of resources. All the typical and atypical scenarios need to be covered, but in doing so, not all threads of execution within the system may be covered. The system may contain internal branches, errors, or interrupts that will lead to untested threads of execution. Tools exists to measure code coverage.

Systems are full of undiscovered bugs. The customer becomes a logical member of the testing team and bug fixes are pushed off to the next release.

2.4 The Implementation Phase

Phase	Deliverable
Implementation	• Code • Critical Error Removal

Table 2.3: The Implementation Phase: Now build it!

In the **implementation phase**, the team builds the components either from scratch or by composition. Given the architecture document from the design phase and the requirement document from the analysis phase, the team should build exactly what has been requested, though there is still room for innovation and flexibility. For example, a component may be narrowly designed for this particular system, or the component may be made more general to satisfy a reusability guideline. [2] The architecture document should give guidance. Sometimes, this guidance is found in the requirement document. The implementation phase is summarized in Table 2.3 on page 18.

The implementation phase deals with issues of quality, performance, baselines, libraries, and debugging. The end deliverable is the product itself.

There are already many established techniques associated with implementation. This thesis does not depend on which technique is followed.

2.4.1 Critical Error Removal

There are three kinds of errors in a system, namely critical errors, non-critical errors, and unknown errors.

[2] A reusability guideline defines a general purpose component that may have many uses across many systems.

A **critical error** prevents the system from fully satisfying the usage scenarios. These errors have to be corrected before the system can be given to a customer or even before future development can progress.

A **non-critical error** is known but the presence of the error does not significantly affect the system's perceived quality. There may indeed be many known errors in the system. Usually these errors are listed in the release notes and have well established work arounds.

In fact, the system is likely to have many, yet-to-be-discovered errors. The effects of these errors are unknown. Some may turn out to be critical while some may be simply fixed by patches or fixed in the next release of the system.

2.5 The Testing Phase

Phase	Deliverable
Testing	• Regression Test • Internal Testing • Unit Testing • Application Testing • Stress Testing

Table 2.4: The Testing Phase: Improve Quality.

Simply stated, quality is very important. Many companies have not learned that quality is important and deliver more claimed functionality but at a lower quality level. It is much easier to explain to a customer why there is a missing feature than to explain to a customer why the product lacks quality. A customer satisfied with the quality of a product will remain loyal and wait for new functionality in the next version. Quality is a distinguishing attribute of a system indicating the degree of excellence.

The testing phase is summarized in Table 2.4 on page 19. For more information on testing see [10], [11], [66], [67], [78], [79], [72], [96], [102], [113], [8], [61], [62], [43], [35], and [113].

In many software engineering methodologies, the **testing phase** is a separate phase which is performed by a different team after the implementation is completed. There is merit in this approach; it is hard to see one's own mistakes, and a fresh eye can discover obvious errors much faster than the person who has read and re-read the material many times. Unfortunately, delegating testing to another team leads to a slack attitude regarding quality by the implementation team.

Alternatively, another approach is to delegate testing to the the whole organization. If the teams are to be known as craftsmen, then the teams should be responsible for establishing high quality across all phases. Sometimes, an attitude change must take place to guarantee quality. [3]

Regardless if testing is done after-the-fact or continuously, testing is usually based on a regression technique split into several major focuses, namely **internal**, **unit**, **application**, and **stress**.

The testing technique is from the perspective of the system provider. Because it is nearly impossible to duplicate every possible customer's environment and because systems are released with yet-to-be-discovered errors, the customer plays an important, though reluctant, role in testing. As will be established later in the thesis, in the WaterSluice methodology this is accomplished in the alpha and beta release of the system.

[3]For some reason, many engineering organizations think that quality assurance is below their dignity. The better attitude would be that every member of an engineering organization should make quality an important aspect.

2.5.1 Regression Test

Quality is usually appraised by a collection of **regression tests** forming a suite of programs that test one or more features of the system.

A regression test is written and the results are generated. If the results are in error, then the offending bug is corrected. A valid regression test generates verified results. These verified results are called the "**gold standard**." This term is borrowed from financial markets where paper money issued by governments was backed by real gold. [4]

Ideally, the validity of a test result is driven by the requirement document; in practice, the implementation team is responsible for validity interpretation.

The tests are collected, as well as their gold-standard results, into a regression test suite. As development continues, more tests are added, while old tests may remain valid. Because of new development, an old test may no longer be valid. If this is the case, the old test results are altered in the "gold standard" to match the current expectations. The test suite is run generating new results. These new results are then compared with the gold-standard results. If they differ, then a potential new fault has entered the system. The fault is corrected and the development continues. This mechanism detects when new development invalidates existing development, and thus prevents the system from regressing into a fault state.

There are four major focuses of regression testing used to assure quality. A summary is found in Table 2.5 on page 22. The discussion follows.

[4] The software engineering methodology WaterSluice will be used to discover the "gold" in a system.

Focus	Note	Team Responsibility
Internal	Make sure all internal, non-customer-visible components work well.	Implementation Team
Unit	Make sure all customer-visible components work well.	Implementation and Design Teams
Application	Make sure the application can complete all scenarios.	Analysis Team
Stress	Run the application in an environment that is more stressful than the target environment.	All Teams

Table 2.5: Categories of Quality

2.5.2 Internal Testing

Internal testing deals with low-level implementation. Here each function or component is tested. This testing is accomplished by the implementation teams. This focus is also called clear-box testing, or sometimes white-box testing, because all details are visible to the test. Internal limits are tested here.

2.5.3 Unit Testing

Unit testing deals with testing a unit as a whole. This would test the interaction of many functions but confine the test within one unit. The exact scope of a unit is left to interpretation. Supporting test code, sometimes called **scaffolding**, may be necessary to support an individual test. This type of testing is driven by the architecture and implementation teams. This focus is also called black-box testing because only the details of the interface are visible to the test. Limits that are global to a unit are tested here.

In the construction industry, scaffolding is a temporary, easy to assemble and dis-assemble, frame placed around a building to facilitate the construction of the building. The construction workers first build the scaffolding and then the building. Later the scaffolding is removed, exposing the completed building. Similarly, in software testing, one particular test may need some supporting software. This software establishes an environment around the test. Only when this environment is established can a correct evaluation of the test take place. The scaffolding software may establish state and values for data structures as well as providing dummy external functions for the test. Different scaffolding software may be needed from one test to another test. Scaffolding software rarely is considered part of the system.

Sometimes the scaffolding software becomes larger than the system software being tested. Usually the scaffolding software is not of the same quality as the system software and frequently is quite fragile. A small change in the test may lead to much larger changes in the scaffolding.

Internal and unit testing can be automated with the help of coverage tools. A coverage tool analyzes the source code and generates a test that will execute every alternative thread of execution. It is still up to the programmer to combine these test into meaningful cases to validate the result of each thread of execution. Typically, the coverage tool is used in a slightly different way. First the coverage tool is used to augment the source by placing informational prints after each line of code. Then the testing suite is executed generating an audit trail. This audit trail is analyzed and reports the percent of the total system code executed during the test suite. If the coverage is high and the untested source lines are of low impact to the system's overall quality, then no more additional tests are required.

2.5.4 Application Testing

Application testing deals with tests for the entire application. This is driven by the scenarios from the analysis team. Application limits and features are tested here.

The application must successfully execute all scenarios before it is ready for general customer availability. After all, the scenarios are a part of the requirement document and measure success. Application testing represents the bulk of the testing done by industry.

Unlike the internal and unit testing, which are programmed, these test are usually driven by scripts that run the system with a collection of parameters and collect results. In the past, these scripts may have been written by hand but in many modern systems this process can be automated.

Most current applications have graphical user interfaces (GUI). Testing a GUI to assure quality becomes a bit of a problem. Most, if not all, GUI systems have event loops. The GUI event loop contains signals for mouse, keyboard, window, and other related events. Associated with each event are the coordinates on the screen of the event. The screen coordinates can be related back to the GUI object and then the event can be serviced. Unfortunately, if some GUI object is positioned at a different location on the screen, then the coordinates change in the event loop. Logically the events at the new coordinates should be associated with the same GUI object. This logical association can be accomplished by giving unique names to all of the GUI objects and providing the unique names as additional information in the events in the event loop. The GUI application reads the next event off of the event loop, locates the GUI object, and services the event.

The events on the event loop are usually generated by human actions such as typing characters, clicking mouse buttons, and moving the cursor. A simple modification to the event loop can journal the events into a file. At a later time, this file could be used to regenerate the events, as if the human was present, and place them on the event loop. The GUI application will respond accordingly.

A tester, using the GUI, now executes a scenario. A journal of the GUI event loop from the scenario is captured. At a later time the scenario can be repeated again and again in an automated fashion. The ability to repeat a test is key to automation and stress testing.

2.5.5 Stress Testing

Stress testing deals with the quality of the application in the environment. The idea is to create an environment more demanding of the application than the application would experience under normal work loads. This is the hardest and most complex category of testing to accomplish and it requires a joint effort from all teams.

A test environment is established with many testing stations. At each station, a script is exercising the system. These scripts are usually based on the regression suite. More and more stations are added, all simultaneous hammering on the system, until the system breaks. The system is repaired and the stress test is repeated until a level of stress is reached that is higher than expected to be present at a customer site.

Race conditions and memory leaks are often found under stress testing. A race condition is a conflict between at least two tests. Each test works correctly when done in isolation. When the two tests are run in parallel, one or both of the tests fail. This is usually due to an incorrectly managed lock.

A memory leak happens when a test leaves allocated memory behind and does not correctly return the memory to the memory allocation scheme. The test seems to run correctly, but after being exercised several times, available memory is reduced until the system fails.

Chapter 3

Methodologies

In this chapter, three major categories of methodologies are presented: sequential, cyclical, and WaterSluice. The sequential and cyclical methodologies, informally known as the waterfall and spiral methodologies, are generic in design and have been simplified to emphasize a key aspect. In a sequential methodology, the four phase of analysis, design, implementation, and testing follow each other sequentially. In a cyclical methodology, the four phase of analysis, design, implementation, and testing are cycled with each cycle generating an incremental contribution to the final system. The WaterSluice is a hybrid borrowing the steady progress of the sequential methodology along with the iterative increments of the cyclical methodology and adds priority and governors to control change.

These three categories of methodologies form a basis for comparison. In the theory chapter, the categories are analyzed in detail. In the survey of methodology chapter, other more established methodologies, are categorized. Performance characteristics of established methodologies can be analyzed based on this categorization.

The computer software industry has introduced a major confusion in terms of naming of methodologies. The **Boehm-waterfall** methodology, analyzed later in this thesis, is most often quoted as a sequential methodology, but the original paper presents it as a cyclical methodology. However, in the greater computer software industry, the term waterfall has come to mean any sequential methodology. This leads to major confusion and hence the introduction, in this thesis, of the sequential classification. Likewise, the **Boehm-spiral** methodology, also analyzed later in this thesis, is most quoted as a cyclical methodology but behaves more like a sequential methodology with many stages. Yet the term spiral has come to mean any cyclical methodology.

3.1 A Sequential Methodology

3.1.1 Introduction

In a **sequential methodology**, informally known as the waterfall, the analysis phase comes first, then the design phase, followed by the implementation phase, and finally by the testing phase. The team that does each phase may be different, and there may be a management decision point at each phase transition. See Figure 3.1 on page 29.

3.1.2 Why It Works

A sequential methodology is successful when the complexity of the system is low and requirements are static. A sequential methodology simply states that first one should think about what is being built, then establish the plan for how it should be built, and then build it with quality. It allows for a software engineering methodology which is in alignment with hardware engineering methods and practices. It forces a discipline process to avoid the pressures of writing code long before it is known what is going

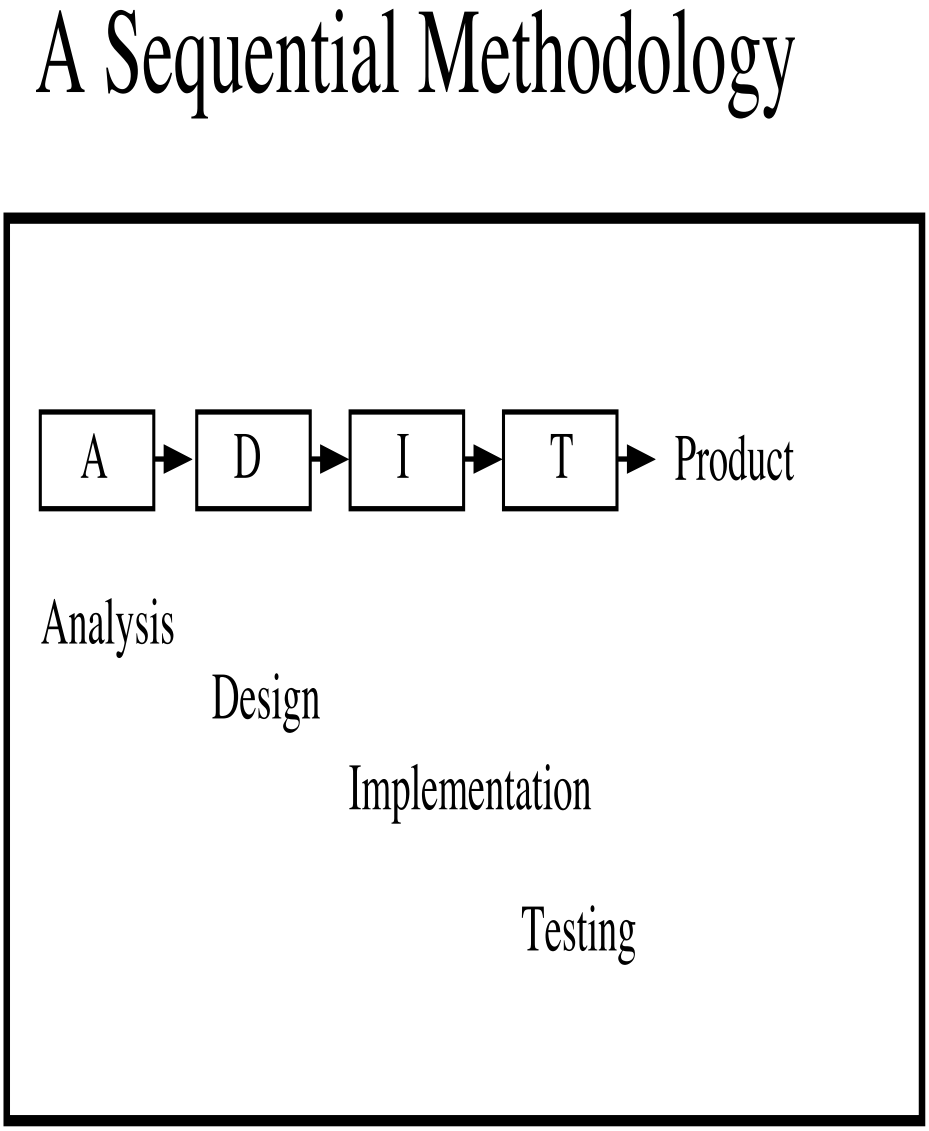


Figure 3.1: A Sequential Methodology

to be built.

Many times, an implementation team is under pressure to build some code before the analysis is completed, only to later discover that the code is not needed or will contribute little to the end product. Unfortunately, this early code becomes a costly legacy: difficult to abandon and difficult to change. A sequential methodology forces analysis and planning before implementation. This is good advice in many software engineering situations.

The process forces the analysis team to precisely define their requirements. It is much easier to build something if it is known what that something is.

Significant numbers of historical software systems have been built using a sequential methodology. Many past corporations owe their success to one of the many sequential methodologies. These successes were in part due to the usage of a formal sequential methodology at the time when pressures of change coming from external sources were limited.

3.1.3 Why It Does Not Work

A sequential methodology might fail for many reasons. A sequential methodology requires the analysis team to be nearly clairvoyant. They must define ALL details up front. There is no room for mistakes and no process for correcting errors after the final requirements are released. There is no feedback about the complexity of delivering code corresponding to each one of the requirements. An easily stated requirement may significantly increase the complexity of the implementation, and it may not even be possible to be implemented with today's technology. Had the requirement team known that a particular requirement could not be implemented, they could have substituted a slightly different requirement that met most of their needs and could have been easier to achieve.

Communication between teams becomes a gating item. Traditionally, the four teams may be different and cross-team communication may be limited. The main mode of communication are the documents that are completed by one team and then passed to another team with little feedback. The requirement team has completed the analysis and is disbanded when the implementation team starts. The requirement documents can only capture a small fraction of the knowledge and typically do not capture any information dealing with quality, performance, behavior, or motivation.

In a fast-moving technology, a sequential methodology builds products that, by the time they are delivered, may be obsolete. A sequential methodology puts so much emphasis on planning, that in a fast-moving target arena, it can not respond fast enough to change. There is no early feedback from the customer and customers may change their requirements. Frequently, once the customers see a prototype of the system, the customers change their requirements.

3.2 A Cyclical Methodology

3.2.1 Introduction

A **cyclical methodology**, informally known as the spiral, fixes some of the problems introduced by a sequential methodology. A cyclical methodology still has the four phases. A little time is initially spent in each phase, followed by several iterations over all four phases.

Simply, the methodology iterates over the processes of think a little, plan a little, implement a little, then test a little. The document structures and deliverable types from each phase incrementally change in structure and content with each cycle or iteration. More detail is generated as the methodology progresses. Finally, after several iterations, the product is complete and ready to ship. The cyclical methodology may continue shipping multiple versions of the product. Ideally, each phase is given equal attention. See Figure 3.2 on page 33.

3.2.2 Why It Works

A cyclical methodology is an incremental improvement on a sequential methodology. It allows for feedback from each team about the complexity of each requirement. There are stages where mistakes in the requirements can be corrected. The customer gets a peek at the results and can feed back information especially important before final product release. The implementation team can feed performance and viability information back to the requirement team and the design team. The product can track technology better. As new advances are made, the design team can incorporate them into the architecture.

3.2.3 Why It Does Not Work

A cyclical methodology has no governors to control oscillations from one cycle to another cycle. Without governors, each cycle generates more work for the next cycle leading to time schedule slips, missing features, or poor quality. More often than not, the length or number of cycles may grow. There are no constraints on the requirement team to "get things right the first time." This leads to sloppy thinking from the requirement team, which gives the implementation team many tasks that eventually get thrown out.

The architecture team is never given a complete picture of the product and hence may not complete a global architecture which scales to full size. There are no firm

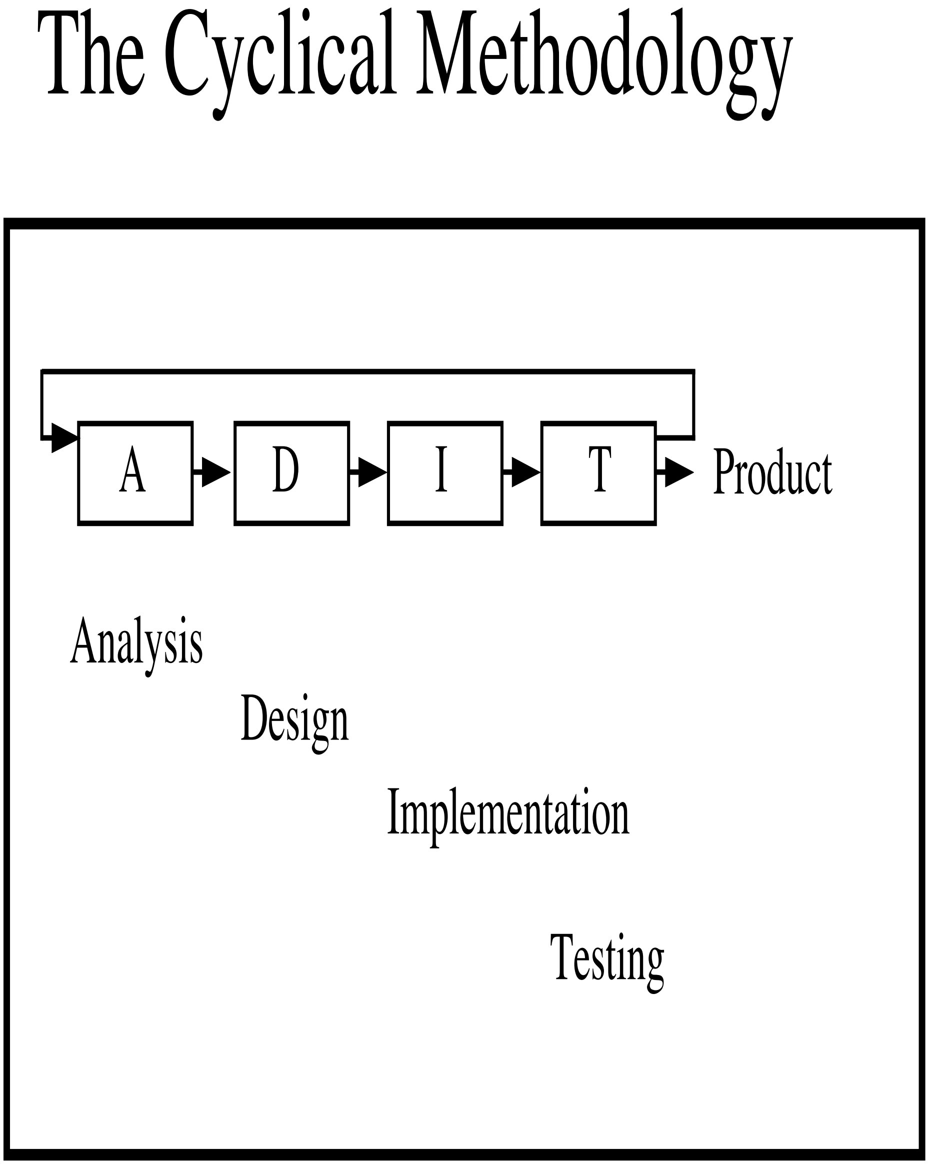


Figure 3.2: A Cyclical Methodology

deadlines. Cycles continue with no clear termination condition. The implementation team may be chasing a continuously changing architecture and changing product requirements.

3.3 The WaterSluice

3.3.1 Introduction

A water sluice is a gold mining technique. Crushed ore and gravel are mixed with fast moving water and then channeled over a long trough with a series of perpendicular-to-the-flow slats. Each row of slats gets smaller as the water flows longer in the channel. Since gold is heavier than the surrounding rock, the gold nuggets collect at these slats. The larger nuggets collect at the bigger slats while the finer specks collect at the smaller slats. The sluice separates the valuable gold from the gravel, concentrating on the big nuggets first. The final product is a smelt of all the nuggets into one gold bar.

See Figure 3.3 on page 35 for a technical diagram of a gold sluice. Picture courtesy of RMS Ross Corporation [112].

Similarly, the **WaterSluice** software engineering methodology separates the important aspects from the less important and concentrates on solving them first. As the process continues, finer and finer details are refined until the product is released. The WaterSluice borrows the iterative nature of a cyclical methodology along with the steady progression of a sequential methodology.

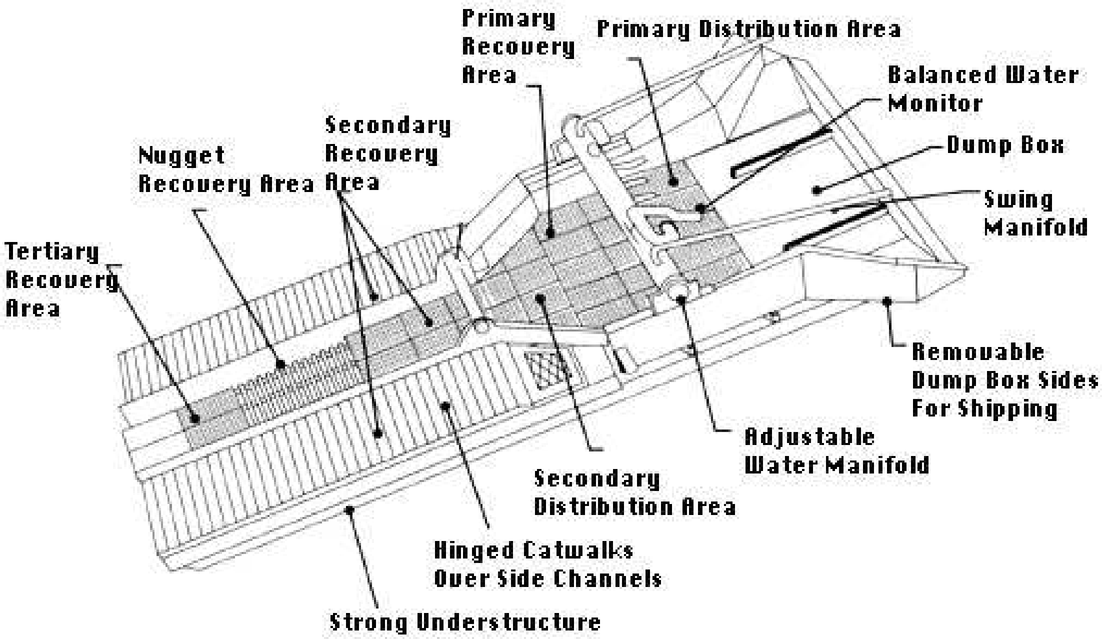


Figure 3.3: A Gold Sluice Diagram

3.3.2 The Process

See Figure 3.4 on page 37 for an overview of the WaterSluice methodology.

Beginning the Process

At the beginning of the project, in an iterative process, the analysis, design, implementation, and test phases are broken into many potential tasks yet to be accomplished by team members. Each potential task is assigned a priority by team members. This priority reflects the benefit to the final goal of accomplishing the task based on what has already been accomplished. The highest priority task is accomplished next. Depending on the size of the team, multiple high priority tasks may be accomplished in parallel. The remaining, lower priority tasks are held for later review. Exactly how many tasks or the granularity of the tasks is dependent on the size of the project, the size of the team building the project, and the scheduled delivery time for the project.

It is important that the decomposition of the problem is done well, regardless of the methodology being used, but especially here in the WaterSluice methodology because priority needs to be accessed. The better the decomposition and priority setting, the more efficient this methodology will perform. More comments on this topic are deferred to a later section. See section C.5 on enabling paradigms located on page 188.

Iterating the Process

As a result of accomplishing these tasks, new analysis, design, implementation, or testing tasks may be discovered. These newly discovered tasks are then added to the known remaining open tasks and again prioritization is required. The next highest priority task are then accomplished.

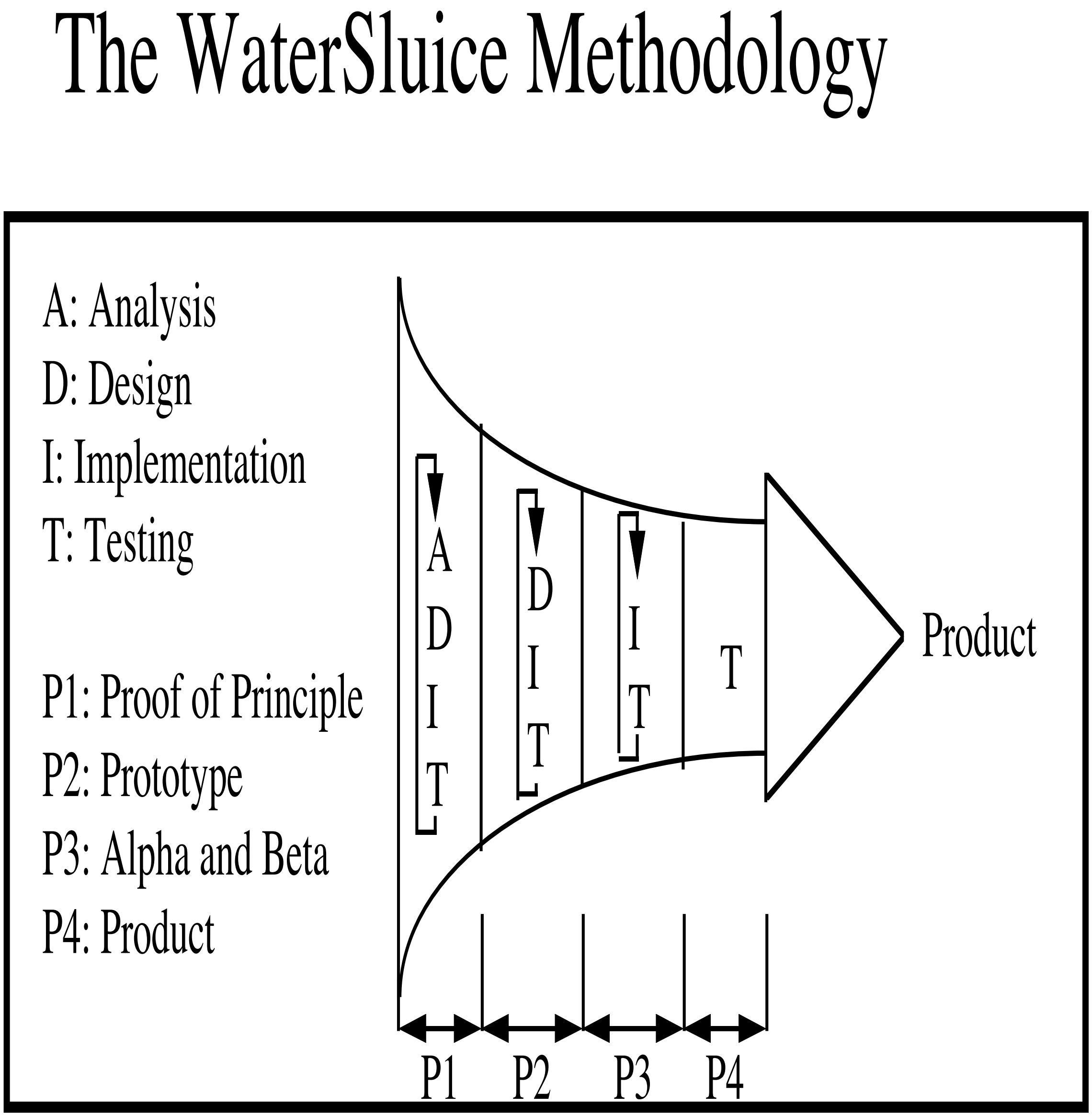


Figure 3.4: The WaterSluice Methodology

Completion of the Process

This process continues until the product is ready for release.

Priority Function

Defining the **priority function** is of high importance. This priority function is domain-specific as well as institution-specific, representing trade-offs between quantity and quality, between functionality and resource constraints, and between expectations and the reality of delivery. The priority function orders the different metrics and their values. However, all priority functions should have the product delivery as a high priority goal. See Appendix D for a discussion on decision making.

The priority function serves two goals. One goal is to establish priority. Important tasks need to be accomplished first over lower priority tasks. This is the traditional role of a priority function.

The second goal of the priority function is to manage conflicting and non-monotonic tasks. The priority function needs to divide the tasks into consistent collections. The priority function needs to guide the selection of the consistent collection and then followed by the selection of the tasks within that consistent selection.

As more and more of the system is established, the priority function is weighted to choose tasks that are consistent with the already established system. A non-monotonic task is inconsistent with the established base requiring that some of the already accomplished system to be thrown out. The non-monotonic task should not be taken, unless the addition of the non-monotonic task is absolutely necessary to the success of the entire system. The priority function guides this decision.

The priority function manages non-monotonic conflicts in the small while, as will be established soon, change order control manages non-monotonic conflicts in the large.

Focus on the Goal

Once a component is completed to the satisfaction of the team, it is placed under **change-order control**. When a component is placed under the change-order control process, changes to the component are now frozen. If a change is absolutely necessary, and the teams are willing to delay the project to enforce the consequences of the change, then the change is fulfilled. Changes should be few, well justified, and documented.

Obviously, early in the process, analysis tasks are naturally a high priority. Later in the process, testing and quality become a higher priority. This is where the change-order control process becomes important. At the beginning of the process all four categories of analysis, design, implementation, and testing are available for prioritizing and scheduling. At the P1-P2 transition point, see Figure 3.4 on page 37, in the process, the analysis phase is subjected to change-order control process. Having the analysis phase frozen focuses attention on the remaining three categories of tasks. In a similar fashion, at the P2-P3 transition point, see Figure 3.4 on page 37, the design phase is frozen and at the P3-P4 transition point the implementation phase is frozen. At the final stage only changes that affect quality are allowed. This leads to a definition of temporal stages in the methodology, specifying priorities.

Don't confuse phases with stages. A phase is a grouping of similar activities. A stage is a temporal grouping of tasks within phases at particular times. Stages follow one another.

Stages

The main **stages** are called **proof-of-principle**, **prototype**, **alpha** and **beta** release, and **product**. With the exception of the proof-of-principle stage, these stages should not be new concepts to software engineers. The proof-of-principle stage represents the more traditional specification stage. Rapid prototyping offers a similar proof-of-principle stage.

Proof-of-Principle Stage In the first stage, the teams work simultaneously on all phases of the problem. The analysis team generates requirements. The design team discusses requirements and feeds back complexity issues to the requirement team and feeds critical implementation tasks to the implementation team. The testing team prepares and develops the testing environment based on the requirements.

The implementation team has to be focused on the critical tasks which is usually the hardest task. This contrasts the common practice of doing the simple things first and waiting until late in the product implementation to tackle the harder tasks. Most products that follow this practice end up failing. Once the critical task components have been implemented, the system, still a child in the first period of life, is ready for transition to the prototype stage.

One of the goals of this stage is for the teams to convince themselves that a solution can be accomplished.

Prototype Stage In the second stage, **the prototype stage**, the requirements and the requirement document are frozen and placed under change-order control. Changes in requirements are still allowed but should be very rare. Any new requirements after this point are very costly. Only if the requirement change is absolutely necessary to the success of the product, despite the potential delays in the product delivery or cost over-runs, is the requirement change allowed. The main idea is to force control on any new requirements. This forces the cycle to be completed and enables product delivery. The architecture is still allowed to vary a little as technology pressures deliver new options.

Once the critical tasks are done well, the implementations associated with the critical tasks are expanded to cover more and more of the application.

One of the goals of this stage is for the team to convince non-team members that the solution can be accomplished.

At the end of this stage, the process is ready for transition into the alpha and beta release stages.

Alpha and Beta Release Stages In the third stage, the architecture is frozen and placed under change-order control. This means that no more architectural changes are allowed unless they are absolutely necessary. Emphasis is now placed on the implementation and quality assurance.

The first version in field release is usually called an **alpha** release, while a second release is called the **beta**. The product may be immature in the alpha release. Only critical tasks have been implemented with high quality. Usually, only a limited number of customers are willing to accept an alpha version of the product and assume the associated risk.

During the beta release, enough of the system should be working to convince the customer that soon the beta application will be a real product. The beta release is more mature and is given to a much larger customer base.

When enough of the system is built, the system is ready for a transition into the next stage: releasing a high quality product.

Product In the fourth stage, the implementation is frozen and focus is primarily on quality. At the end of the stage, the **product** is delivered.

One of the goals of the last stage is to make the product sound and of high quality. No known critical errors are allowed in the final product. Sometimes, there is a gray area of definition between a product feature and a product error with the provider of the product, most often then not, providing features, while the customers viewing some features as errors.

The process is then repeated for the next version of the product.

The WaterSluice allows for phase interactions while at the same time setting firm temporal deadlines. The WaterSluice forces all four phases to communicate up front and to work together.

The WaterSluice software engineering methodology assumes the presence of five levels in a supporting software engineering environment as described in the appendix. Versioning is used to move the product from one version to another version by repeating the methodology for each version. Risk management is assumed throughout the process. The major components of analysis, the details in the design phase, the four main phases of implementation, and levels of testing proceed as previously described.

Change-Order Control

Change-order control is a software engineering process that manages change, or lack there of. The process is weighted to prevent change. Tools help to manage this process, while senior decision makers accept or decline change decisions. Frequently, the senior decision makers are independent of the teams.

Once a component is completed to the satisfaction of the team, it is placed under change-order control. When a component is placed under the change-order control process, changes to the component are now frozen. If a change is absolutely necessary, and the senior decision makers are willing to delay the project to enforce the consequences of the change, then the change is fulfilled. Changes should be few, well justified, and documented.

Many change requests are postponed and incorporated into the next version of the product. Some of these change requests contribute to the requirement document for the next version, while some contribute to the architecture and implementation. Still, others may improve the quality.

3.3.3 Why It Works

There are many things that work well in the WaterSluice methodology. The WaterSluice methodology recognizes that people make mistakes and no decision can be absolute. The teams are not locked into a requirement or an architecture decision that turns out to be wrong or no longer appropriate. The methodology forces explicit freeze dates. This allows for the product to be built and shipped. It forces accountability by having decision points where, for the most part, things need to be completed. The first stage is iterative allowing for the correction of mistakes. Even after a portion of the system goes under change-order control, a decision can be changed if it is absolutely necessary.

The WaterSluice methodology forces the teams to think but does not require the teams to be clairvoyant. Sufficient time is allowed for the first stage to establish the confidence level needed for success. Communication is emphasized.

The WaterSluice methodology allows for fast interaction, up front, between all phases of analysis, design, implementation, and testing. This feeds critical information between all four phases. The implementation team doesn't waste time working on throw-away code because requirements are validated early in the process for feasibility of implementation.

The WaterSluice methodology can respond to market changes more quickly due to the iterative nature in each stage allowing requirements to enter and exit at each stage. The WaterSluice methodology tries to move all mistakes to the beginning of the process, where a restart is not very costly.

3.3.4 Why It Does Not Work

The WaterSluice methodology forces accountability by having clearly defined stages where activities are frozen and placed under change order control. Many people are not willing to take that responsibility. For the WaterSluice methodology to work, it is necessary to create an environment where taking responsibility and accountability for a decision need not be detrimental to the individual if the decision later leads to a failure. Otherwise, people will avoid accepting accountability, leading to missed goals.

An attitude change towards testing is necessary by the teams since all teams are involved in testing from the beginning. The WaterSluice methodology requires that people communicate well up front, which is difficult since all four phases represent different perspectives. The methodology trades off total flexibility with the reality of product delivery.

3.4 Conclusion

In this chapter, three major categories of methodologies were presented: sequential, cyclical, and WaterSluice. The sequential and cyclical methodologies, informally known as the waterfall and spiral methodologies, are generic in design and have been simplified to emphasize a key aspect. In a sequential methodology, the four phases of analysis, design, implementation, and testing follow each other sequentially. In a cyclical methodology, the four phases of analysis, design, implementation, and testing are cycled with each cycle generating an incremental contribution to the final system. The WaterSluice is a hybrid borrowing the steady progress of the sequential methodology along with the iterative increments of the cyclical methodology and adds priority and governors to control change.

A sequential methodology is successful when the complexity of the system is low and requirements are static. In a fast-moving technology, a sequential methodology builds products that, by the time they are delivered, may be obsolete. A sequential methodology puts so much emphasis on planning, that in a fast-moving target arena, it can not respond fast enough to change.

A cyclical methodology is an incremental improvement on a sequential methodology, allowing for incremental feedback between cycles. A cyclical methodology has no governors to control oscillations from one cycle to another cycle. Without governors, each cycle may generates more work for the next cycle.

The WaterSluice methodology introduced priority, goal-focus, and change-control management. A system moves through the states of proof-of-principle, prototype, alpha and beta release, and product. In a later chapter it will be shown that a software engineering methodology that is goal focused, manages conflicts, and differentiates between different priorities is best suited for dynamic non-monotonic environments.

Chapter 4

Established Methodologies

There are many established software engineering methodologies. This section concentrates on several established software engineering methodologies with emphasis on the two most known, specifically the Boehm-Waterfall and the Boehm-Spiral methodology.

4.1 The Boehm-Waterfall Methodology

The Boehm-Waterfall software engineering methodology [20] is one of the best known example of a software engineering methodology. The Boehm-Waterfall software engineering methodology is composed into the stages of system requirements, software requirements, preliminary and detailed design, implementation, testing, operations, and maintenance. At each stage is a validation step. In the Boehm-Waterfall software engineering methodology, as often quoted and viewed, the process flows from stage to stage like water over a fall. However, in the original description of the Boehm-Waterfall software engineering methodology, there is an interactive backstep between each stage. Thus the Boehm-Waterfall is a combination of a sequential methodology with an interactive backstep. However, in engineering practice, the term waterfall is used as a generic name to any sequential software engineering methodology.

See Figure 4.1 on page 48 from [20].

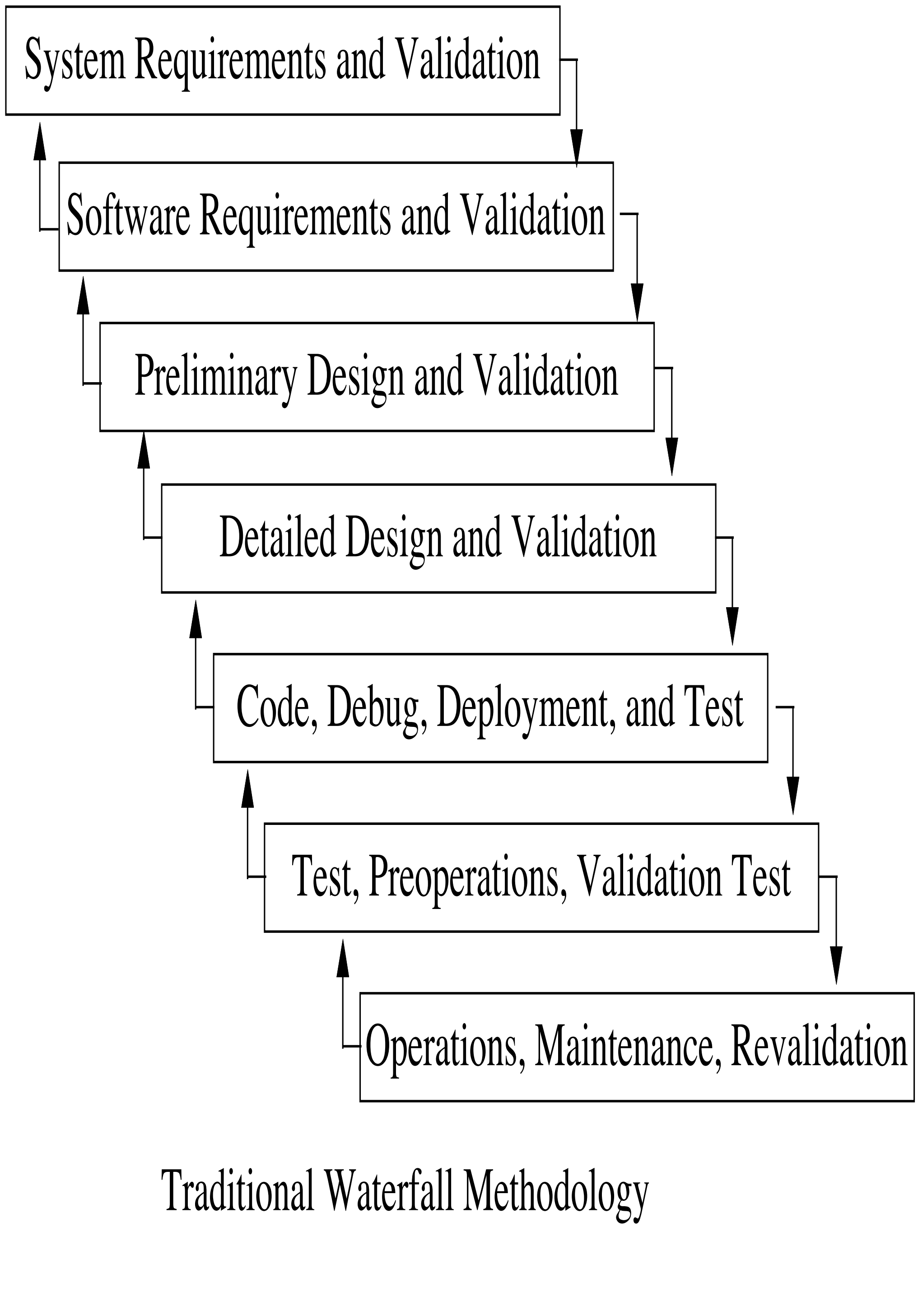


Figure 4.1: The Boehm-Waterfall Methodology

4.2 The Boehm-Spiral Methodology

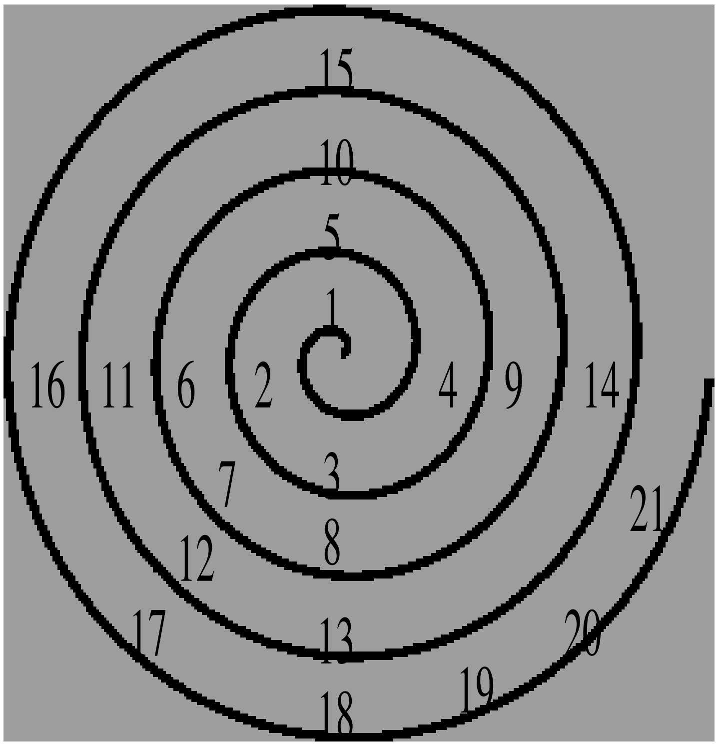
The Boehm-Spiral software engineering methodology spiral [22] is a well another known examples of a software engineering methodology. See Figure 4.2 on page 50 from [22].

The Boehm-Spiral software engineering methodology is composed into many stages. See Table 4.1 on page 49.

Cycle	Step
Cycle 1 - Early Analysis	• Step 1: Objectives, Alternatives, and Constraints • Step 2: Risk Analysis and Prototype • Step 3: Concept of Operation • Step 4: Requirement and Life cycle Plan • Step 5: Objectives, Alternatives, and Constraints • Step 6: Risk Analysis and Prototype
Cycle 2 - Final Analysis	• Step 7: Simulation, Models, and Benchmarks • Step 8: Software Requirements and Validation • Step 9: Development Plan • Step 10: Objectives, Alternatives, and Constraints • Step 11: Risk Analysis and Prototype
Cycle 3 - Design	• Step 12: Simulation, Models, and Benchmarks • Step 13: Software Product Design, Validation, and Verification • Step 14: Integration and Test Plan • Step 15: Objectives, Alternatives, and Constraints • Step 16: Risk Analysis and Operational Prototype
Cycle 4 - Implementation and Testing	• Step 17: Simulation, Models, and Benchmarks • Step 18: Detailed Design • Step 19: Code • Step 20: Unit, Integration, and Acceptance Testing • Step 21: Implementation (Deployment)

Table 4.1: Boehm-Spiral Methodology Stages

The processes starts in the center of the spiral. Each completed cycle along



Traditional
Spiral
Methodology

1 Objectives, Alternatives, and Constraints
2 Risk Analysis and Prototype
3 Concept of Operation
4 Requirement and Life-cycle Plan
5 Objectives, Alternatives, and Constraints
6 Risk Analysis and Prototype
7 Simulation, Models, and Benchmarks
8 Software Requirements and Validation
9 Development Plan
10 Objectives, Alternatives, and Constraints
11 Risk Analysis and Prototype
12 Simulation, Models, and Benchmarks
13 Software Product Design, Validation, and Verification
14 Integration and Test Plan
15 Objectives, Alternatives, and Constraints
16 Risk Analysis and Operational Prototype
17 Simulation, Models, and Benchmarks
18 Detailed Design
19 Code
20 Unit, Integration, and Acceptance Testing
21 Implementation (Deployment)

Figure 4.2: The Boehm-Spiral Methodology

the spiral represents one stage of the process. As the spiral continues, the product matures.

In the Boehm-Spiral software engineering methodology, as often quoted and viewed, the process spirals from stage to stage, with each spiral getting closer and closer to a final solution. However, the Boehm-Spiral software engineering methodology also has a steady progress from one stage into the next stage with an explicit review between each stage. Thus the Boehm-Spiral is a hybrid of both a sequential and a cyclical software engineering methodology. However, in engineering practice, the term spiral is used as a generic name to any cyclical software engineering methodology, including cycles leading to prototypes and multiple versions.

4.3 Versions

Another important software engineering methodology is versioning where the system development is broken down into a series of smaller goals. The system is released in a series of versions with each version potentially adding more functionality. Using versions develops the system in a sequential manor while, if viewed from the software engineering life cycle prospective, a more cyclical approach is taken. Thus, versioning is a hybrid of both a sequential and a cyclical software engineering methodology. See Figure 4.3 on page 52.

Each version replays the methodology. Frequently the previous version becomes the starting point for the next version. Some features may be deferred to a later version. Changes in the requirements that happen in the design or implementation phase are usually deferred to a later version. The selection of features in any one version is a complex process involving resource constraints, customer requirements,

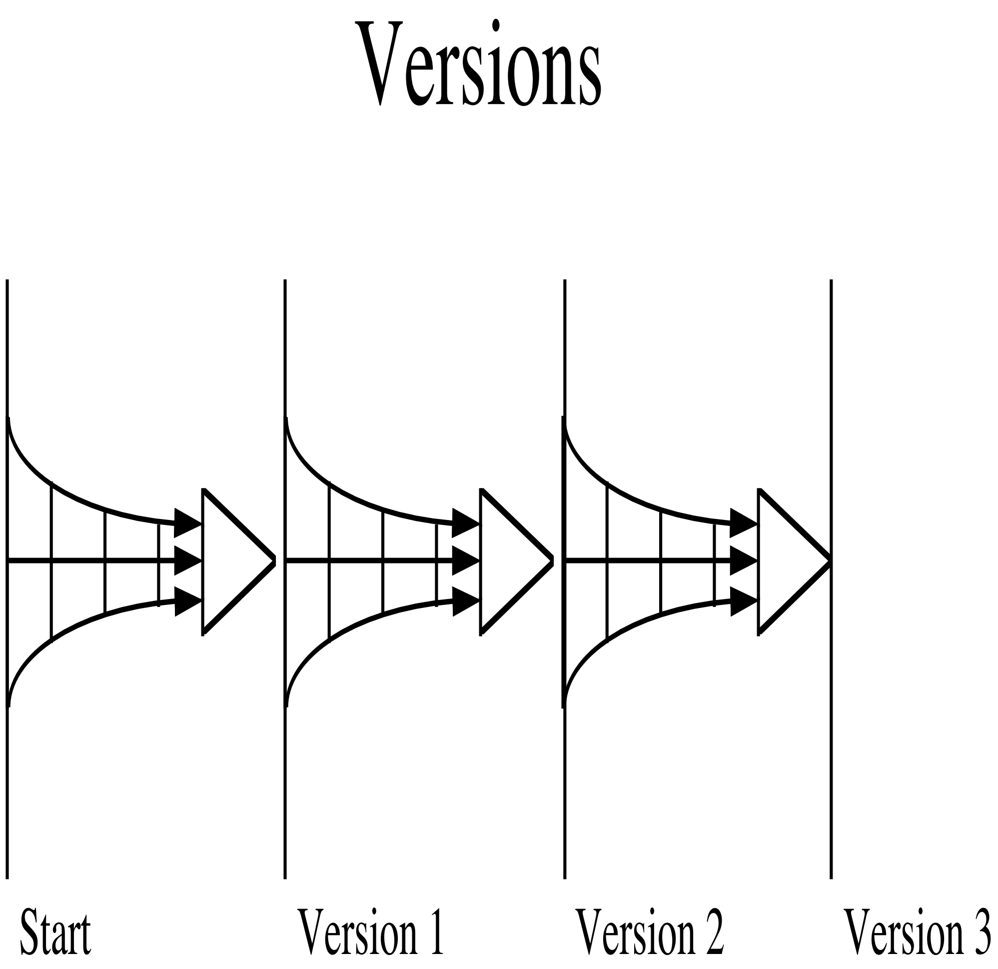


Figure 4.3: The Version Process

availability of support environments, and availability of skilled people.

Many times, one released version concentrates on quality improvements while the following release version concentrates on added functionality. This alternating release schedule is common and reflects the difficulty in reaching high product quality before product visibility. First the required functionality is released in a version and then, after the customers use the version, a new version is released with the newly discovered errors fixed. The environments in which the customers use the product may be so variable as to preclude exhaustive testing. Of course, exhaustive testing is seldom accomplished in any one product.

4.4 The Booch Methodology

The Booch software engineering methodology [26] provides an object-oriented development in the analysis and design phases. The analysis phase is split into steps. The first step is to establish the requirements from the customer perspective. This analysis step generates a high-level description of the system's function and structure. The second step is a domain analysis. The domain analysis is accomplished by defining object classes; their attributes, inheritance, and methods. State diagrams for the objects are then established. The analysis phase is completed with a validation step. The analysis phase iterates between the customer's requirements step, the domain analysis step, and the validation step until consistency is reached.

Once the analysis phase is completed, the Booch software engineering methodology develops the architecture in the design phase. The design phase is iterative. A logic design is mapped to a physical design where details of execution threads, processes, performance, location, data types, data structures, visibility, and distribution are established. A prototype is created and tested. The process iterates between the logical design, physical design, prototypes, and testing.

The Booch software engineering methodology is sequential in the sense that the analysis phase is completed and then the design phase is completed. The methodology is cyclical in the sense that each phase is composed of smaller cyclical steps. There is no explicit priority setting nor a non-monotonic control mechanism. The Booch methodology concentrates on the analysis and design phase and does not consider the implementation or the testing phase in much detail.

4.5 Object Modeling Technique (OMT)

The Object Modeling Technique (OMT) software engineering methodology [117] is another well known example of a software engineering methodology. The OMT software engineering methodology deals with object-oriented development in the analysis and design phases.

The analysis phase starts with a problem statement which includes a list of goals and a definitive enumeration of key concepts within a domain. This problem statement is then expanded into three views, or models: an object model, a dynamic model, and a functional model. The object model represents the artifacts of the system. The dynamic model represents the interaction between these artifacts represented as events, states, and transitions. The functional model represents the methods of the system from the perspective of data flow. The analysis phase generates object-model diagrams, state diagrams, event-flow diagrams, and data-flow diagrams. The analysis phase is now complete.

The system design phase follows the analysis phase. Here the overall architecture is established. First the system is organized into subsystems which are then allocated to processes and tasks, taking into account concurrency and collaboration. Then persistent data storage is established along with a strategy to manage shared-global information. Next, boundary situations are examined to help guide trade-off priorities.

The object design phase follows the system design phase. Here the implementation plan is established. Object classes are established along with their algorithms with special attention to the optimization of the path to persistent data. Issues of inheritance, associations, aggregation, and default values are examined.

The OMT software engineering methodology is sequential in the sense that first comes analysis, followed by design. In each phase, a cyclical approach is taken among the smaller steps. The OMT is very much like the Booch methodology where emphasis is placed on the analysis and design phases for initial product delivery. Both the OMT and Booch do not emphasize implementation, testing, or other life cycle stages.

4.6 Rational Objectory Methodology

The Rational Objectory [86], [73] is a full life cycle software engineering methodology. Rational Objectory is an iterative process governed by requirements management. Rational Objectory activities create and maintain models to aid the developer in supporting the methodology.

The Rational Objectory software engineering methodology can be described in two dimensions: time and process components. The time dimension represents the dynamic aspect of the process and is expressed in terms of cycles, phases, iterations

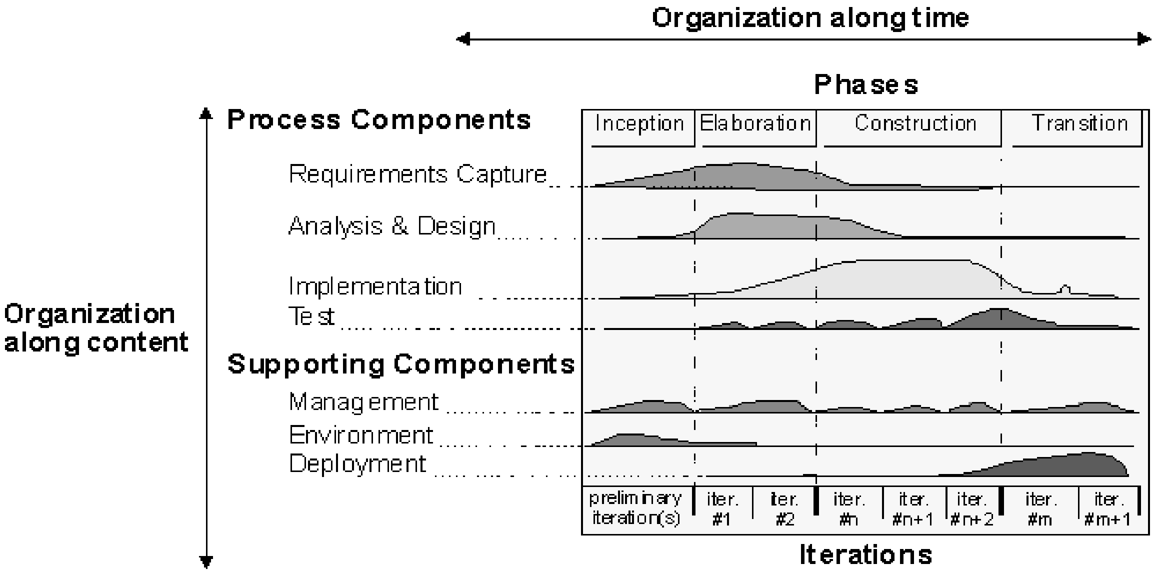


Figure 4.4: The Rational Objectory Methodology.

and milestones. The process component dimension is described in terms of process components, activities, workflows, artifacts, and workers. See Figure 4.4 on page 56 from [86].

4.6.1 Phases

The software life cycle is broken into cycles with each cycle working on a generation of the system. The Rational Objectory software engineering methodology divides one development cycle into four consecutive phases: inception phase, elaboration phase, construction phase, and transition phase. [1]

[1]The temporal ordering of the inception phase, the elaboration phase, the construction phase, and the transition phase in the Rational Objectory corresponds to the proof-of-principle, prototype, alpha, beta, and product release stages of the WaterSluice.

The Inception Phase

The inception phase establishes the business case for the system and define the system's scope. The business case includes success criteria, risk assessment, estimate of the resources needed, and a phase plan showing dates of major milestones. At the end of the inception phase, the life cycle objectives of the project are examined to decide whether or not to proceed with the development.

Elaboration Phase

The goals of the elaboration phase are to analyze the problem domain, establish a sound architectural foundation, develop the project plan and eliminate the highest risk elements of the project. At the end of the elaboration phase, the detailed system objectives, scope, choice of an architecture, and the resolution of major risks are examined.

The Construction Phase

During the construction phase, a complete system is iteratively and incrementally developed and made ready for transition to the customer community. This includes completing the implementation and testing of the software. At the end of the construction phase, the operational decision is made.

The Transition Phase

During the transition phase, the software is shipped to the customer. This phase typically starts with a "beta release" of the systems. At the end of the transition phase, the life cycle objectives are reviewed and possibly another development cycle begins.

4.6.2 Iterations

Each phase in the Rational Objectory software engineering methodology can be further broken down into iterations. An iteration is a complete development loop resulting in a internal or external system release. Each iteration goes through all aspects of software development: requirement capture, analysis and design, implementation and testing. [2]

The Requirements Capture

The requirements capture process describes what the system should do. Requirements capture results in a use-case model. The use-case model consists of actors and use-cases. Actors represent the customers or another software system. Use-cases represent the behavior of the system. The use-case description shows how the system interacts step-by-step with the actors. The use-cases function as a unifying thread throughout the system's development cycle. The same use-case model is used during requirements capture, analysis and design, and test.

Analysis and Design

The analysis and design process describes an architecture that serves as an abstraction of the source code and a "blueprint" of how the system is structured. The architecture consists of design classes and views. These views capture the major structural design decisions. In essence, architectural views are abstractions or simplifications of the entire design.

[2]The iteration of the fundamental phases of analysis, design, implementation, and testing are the same as in the WaterSluice methodology though with slightly different emphasis.

Implementation

The system is built during implementation. This includes source-code files, header files, make files, and binaries.

Testing

Testing verifies the entire system. Testing includes system level and scenario based tests.

4.6.3 Comparison to WaterSluice

The Rational Objectory software engineering methodology is very similar to the WaterSluice software engineering methodology. The temporal ordering of the inception, elaboration, construction, and the transition phase in the Rational Objectory corresponds to the proof-of-principle, prototype, alpha, beta, and product release stages of the WaterSluice. The fundamental phases of analysis, design, implementation, and testing are the same in the two software engineering methodologies. The importance of each phase in each stage is very similar in the two software engineering methodologies. Rational Objectory does not have an explicit priority function nor a process to manage non-monotonic requirements like the WaterSluice.

4.7 WinWin Spiral Methodology

The WinWin spiral software engineering methodology [25] is a recent example of a software engineering methodology. The WinWin spiral software engineering methodology expands the Boehm-Spiral methodology by adding a priority setting step, the

WinWin process, at the beginning of each spiral cycle and by introducing intermediate goals, called anchor points.

The WinWin process identifies a decision point. For each decision point, the objectives, constraints, and alternatives are established and a WinWin condition is established. This may require a negotiation among the stakeholders and some reconciliations.

The anchor points establish three intermediate goals. The first anchor point, called the life cycle objective (LCO), establishes sound business cases for the entire system by showing that there is at least one feasible architecture that satisfies the goals of the system. The first intermediate goal is established when the top-level system objectives and scope, the operational concepts, the top-level system requirements, architecture, life cycle model, and system prototype are completed. This first anchor point establishes the why, what, when, who, where, how, and cost of the system. At the completion of this anchor point, a high level analysis of the system is available.

The second anchor point, called the life cycle architecture (LCA), defines the life cycle architecture. The third anchor point, called the initial operational capability (IOC), defines the operational capability, including the software environment needed for the first product release, operational hardware and site environment, and customer manuals and training. These two anchor points expand the high level analysis into other life cycle stages.

The WinWin spiral software engineering methodology is similar to the WaterSluice. The WinWin process could be considered a WaterSluice priority function, while the anchor points could represent WaterSluice stages. The WinWin process does not explicitly include non-monotonic effects. The anchor points are like the major stages in the life cycle of a product: initial development, deployment, operations, maintenance, legacy, and final discontinuation. The first anchor point is close to initial development. The second anchor point initiates deployment, while the third anchor point starts operations and maintenance.

4.8 Conclusion

These methodologies reviewed in this chapter, are used in today's software engineering practice and appear to have a positive benefit. They are a considerable improvement on not using any methodology at all [99]. The implementors in any case have much freedom in terms of thoroughness and tool use. The scale of the issue is such that outside of small experiments [105] reliable quantitative measurements of alternative methodologies have not been possible.

Chapter 5

Formal Foundations

In this chapter, the formal foundations are presented including the main theorem of the thesis. In support of the proof of the main theorem, a series of definitions are presented followed by a series of secondary theorems and their corollaries with their associated proofs. Results are then summarized.

5.1 A Preview of the Main Theorem

Theorem 1 *Different software engineering methodologies have significant performance variations depending on the given environment. A software engineering methodology that is goal focused, manages conflicts, and differentiates between different priorities is best suited for dynamic non-monotonic environments.*

To prove this theorem, formal definitions of software engineering methodologies, performance, and environment are now presented. The variations in performance of different software engineering methodologies are sufficiently great as to make the choice of which software engineering methodology to use dependent on the surrounding environment.

5.2 Definitions

5.2.1 Towards the Definition of Environment

First the environment will be defined. As soon discussed, the environment definition is built from the definitions of the analysis, design, implementation, and testing phases. Each phase defines a plane which is then defined in terms of atomic, compound, and complex steps that may have a sibling relationship. Together the four planes form a multi-layered space, either static or dynamic. In some cases, a dynamic space may exhibit the non-monotonic property. There are two special steps: the problem statement and the system acceptance test. The environment is the multi-layered finite space consisting of the analysis, design, implementation, and testing planes with two special steps: the initial problem statement and the system acceptance test.

Definition 1 (Analysis) *The analysis phase defines the requirements of the system in a declarative fashion, independent of how these requirements will be accomplished.*

Section 2.2 defined the analysis phase. In summary, the analysis phase defines the problem that the customer is trying to solve. The deliverable result at the end of the analysis phase is a requirement document. Ideally, the requirement document states in a clear and precise fashion what is to be built. The analysis phase represents the "what" phase. The requirement document tries to capture the requirements from the customer's perspective by defining goals and interactions at a level removed from the implementation details. The analysis phase was summarized in Table 2.1 on page 6. The analysis phase builds a declarative model of the system.

Definition 2 (Design) *The design phase establishes the architecture.*

Section 2.3 defines the design phase. In summary, the design phase starts with the requirement document delivered by the analysis phase and maps the requirements into an architecture. The architecture defines the components of the software system, their interfaces and behaviors. The deliverable design document is the architecture specification. The design document describes a plan to implement the requirements. This phase represents the "how" phase. Details on computer programming languages and environments, machines, packages, application architecture, distributed architecture layering, memory size, platform, algorithms, data structures, global type definitions, interfaces, and many other engineering details are established. The design may include the reuse of existing components. The design phase is summarized in Table 2.2 on page 12. The architecture is a high level mapping of the declarative model of the system into the imperative model defined by the implementation.

Definition 3 (Implementation) *In the implementation phase, the system is built.*

Section 2.4 defines the implementation phase. In summary, in the implementation phase the system is built, performance is enhanced, reusable libraries are established, and errors are corrected. The end deliverable is the product itself. In the implementation phase the team builds the components either from scratch or by composition. Given the architecture document from the design phase and the requirement document from the analysis phase, the team should build what has been requested, though there is still room for flexibility. The implementation phase represents an imperative model of the system

Definition 4 (Testing) *The testing phase improves quality.*

Section 2.5 defines the testing phase. Testing is usually based on the regression paradigm where current results from a test suite are compared to a gold standard. As the testing suite grows the coverage of the system improves and enhances the quality. Testing includes internal testing, unit testing, application testing, and stress testing. The testing phase is summarized in Table 2.4 on page 19.

Definition 5 (Step) *In each of the four phases of analysis, design, implementation, and testing there are many steps.*

Let $a_1, a_2, a_3, \ldots, a_{n_a}$ be the n_a requirement steps leading to a possible analysis A. Let $d_1, d_2, d_3, \ldots, d_{n_d}$ be the n_d architecture steps leading to a possible design D. Let $i_1, i_2, i_3, \ldots, i_{n_i}$ be the n_i implementation steps leading to a possible implementation I. Let $t_1, t_2, t_3, \ldots, t_{n_t}$ be the n_t testing steps leading to a possible testing T. A step is recursively defined in terms of atomic, compound, and complex steps. A step may have sibling relationships with other steps.

Definition 6 (Atomic Step) *An atomic step represents the simplest step with no further decomposition.*

Let a_j be an atomic analysis step then a_j has no decomposition. Let d_j be an atomic design step then d_j has no decomposition. Let i_j be an atomic analysis step then i_j has no decomposition. Let t_j be an atomic analysis step then t_j has no decomposition. Each individual step in the analysis, design, implementation, or testing phases may be an atomic step.

Definition 7 (Compound Step) *A compound step consists of several steps from the same layer organized as a hierarchy, or in the most general case, a directed acyclic graph (DAG).*

Each individual step in the analysis, design, implementation, or testing phases may be a compound step. A compound step can be decomposed into several steps using refinement techniques. A compound step may itself be composed of multiple atomic, compound, or complex steps. See Figure 5.1 on page 67 for a visual representation of a compound step.

Let $a_{j_1}, a_{j_2}, \ldots, a_{j_{n_{a_j}}}$ be the n_{a_j} compound requirement steps leading to a possible analysis a_j. Let $d_{j_1}, d_{j_2}, \ldots, d_{j_{n_{d_j}}}$ be the n_{d_j} compound architecture steps leading to a possible design element d_j. Let $i_{j_1}, i_{j_2}, \ldots, i_{j_{n_{i_j}}}$ be the n_{i_j} compound implementation steps leading to a possible implementation component i_j. Let $t_{j_1}, t_{j_2}, \ldots, t_{j_{n_{t_j}}}$ be the n_{t_j} compound testing steps leading to a possible testing step t_j.

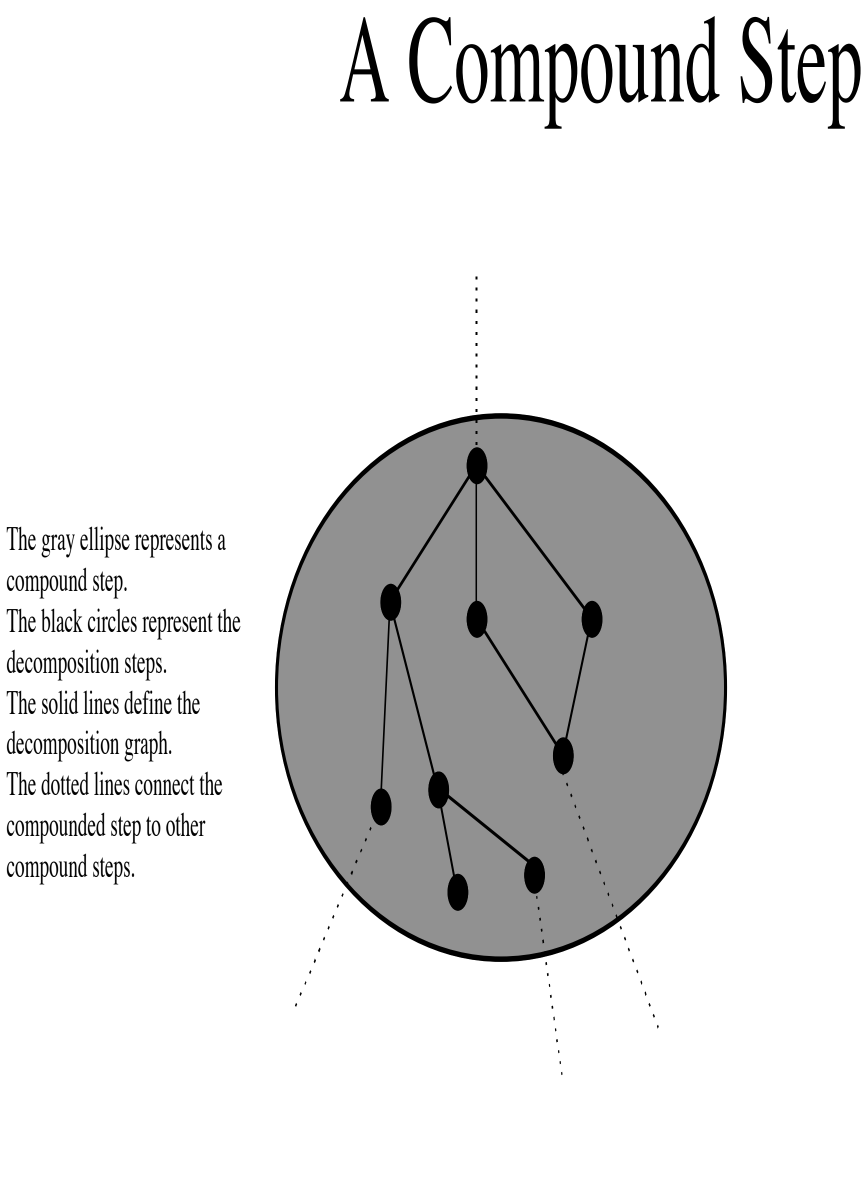


Figure 5.1: A Compound Step

Definition 8 (Complex Step) *A complex step consists of several steps from different layers organized as a hierarchy, or in the most general case, a directed acyclic graph (DAG).*

Steps in the analysis, design, or implementation phases may together form a complex step. In the special case of a testing phase step, the step may be atomic or compound but not complex. An analysis step expands into one or more design steps. A design step expands into one or more implementation steps. While an implementation step expands into one or more testing steps. These expansions include steps to define the parent step as well as sibling steps to support the expansion. The expansions may not be disjoint with other expansion and overlap forming a directed acyclic graph. See Figure 5.2 on page 69 for a visual representation of an example directed acyclic graph complex step consisting of analysis, design, implementation, and testing steps.

Let $d_{j_1}, d_{j_2}, \ldots, d_{j m_{d_j}}$ be the m_{d_j} design steps leading to a possible analysis step a_j. Let $i_{j_1}, i_{j_2}, \ldots, i_{j m_{i_j}}$ be the m_{i_j} implementation steps leading to a possible design step d_j. Let $t_{j_1}, t_{j_2}, \ldots, t_{j m_{i_j}}$ be the m_{i_j} testing steps leading to a possible implementation step i_j.

Definition 9 (Sibling Step Relationship) *Two steps have a sibling relationship if the two steps have overlapping decomposition and do share a common parent in the same layer. Typically, the sibling step is introduced after the decomposition of an existing step.*

The presence of one step may require the inclusion of several sibling steps. A sibling step supports the accomplishment of another sibling step but was not explicitly in the decomposition of the parent compound step. For example, steps derived directly

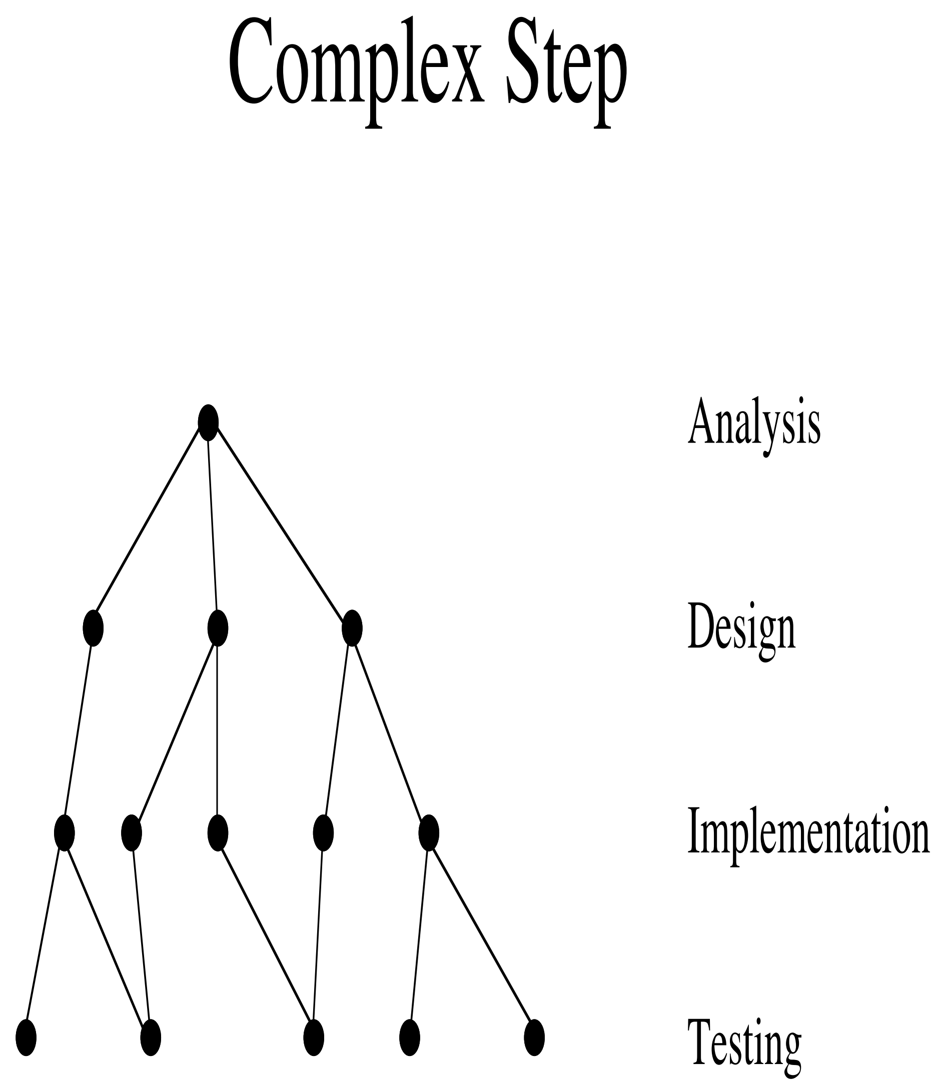


Figure 5.2: A Complex Step

from the problem statement fulfill functional requirements. A chosen architecture in the design phase that supports the functional requirements introduces non-functional requirements. A sibling step fulfills these non-functional requirements. See Figure 5.3 on page 71 for a visual representation of an example sibling relationship.

Definition 10 (Multi-layered Space) *The analysis, design, implementation, and testing steps, either atomic, compound, or complex, form planes that define a multi-layered finite space.*

The n_a analysis steps $a_1, a_2, a_3, \ldots, a_{n_a}$ leading to a possible analysis A form an analysis plane. The n_d design steps $d_1, d_2, d_3, \ldots, d_{n_d}$ leading to a possible design D form a design plane. The n_i implementation steps $i_1, i_2, i_3, \ldots, i_{n_i}$ leading to a possible implementation I form an implementation plane. The n_t testing steps $t_1, t_2, t_3, \ldots, t_{n_t}$ leading to a possible testing suite T form a testing plane.

Let the space S be represented as $< A, D, I, T >$ where A is a possible analysis of n_a steps, D is a possible analysis of n_d steps, I is a possible analysis of n_i steps, and T is a possible analysis of n_t steps. The total number of steps in space S is $n_a + n_d + n_i + n_t$. This space is finite and bounded because we are dealing with the software engineering of only finite and bounded systems. See Figure 5.4 on page 72 for a visual representation of the multi-layered space.

Definition 11 (Static Space) *A static space does not change over time.*

In a static space, all steps are known before any analysis, design, implementation, or testing begins. No new steps enter the space. No existing step leaves the space and no step is in conflict with any other existing step.

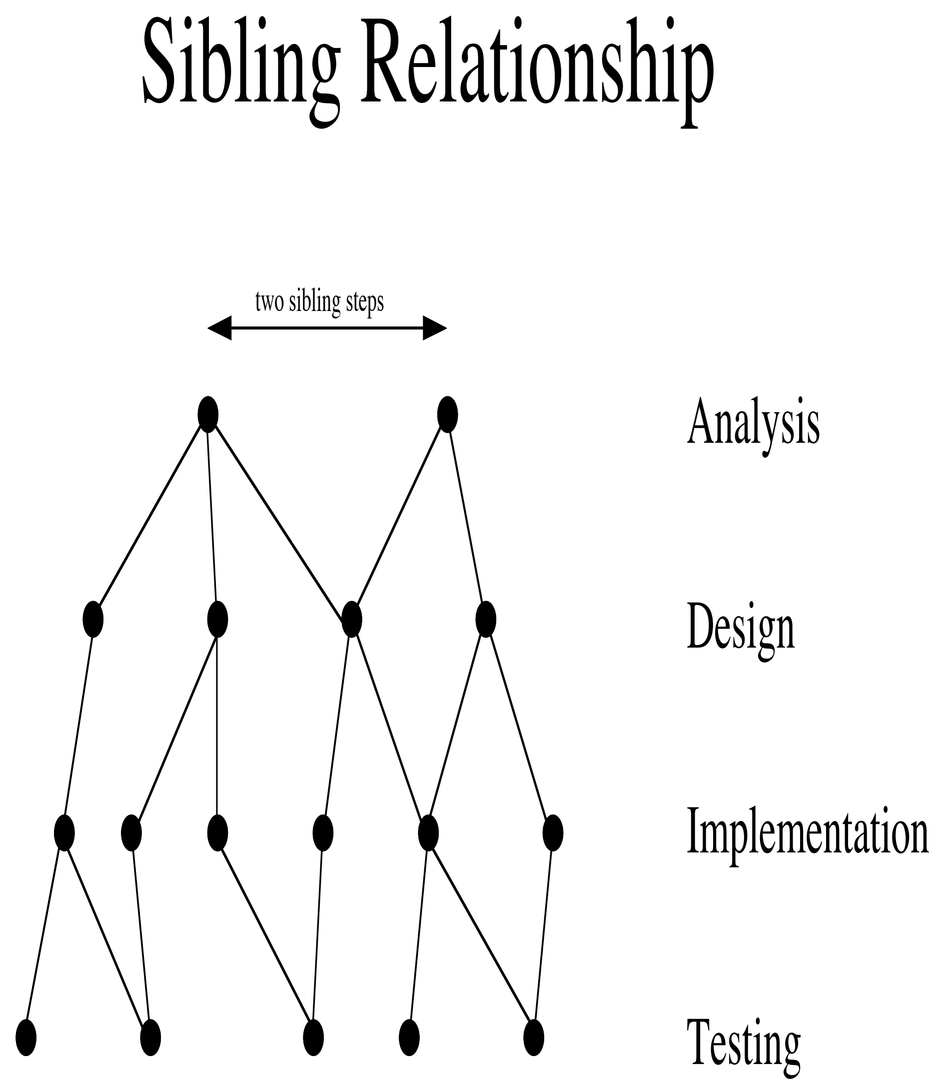


Figure 5.3: Two Sibling Steps and their Overlapping Decomposition

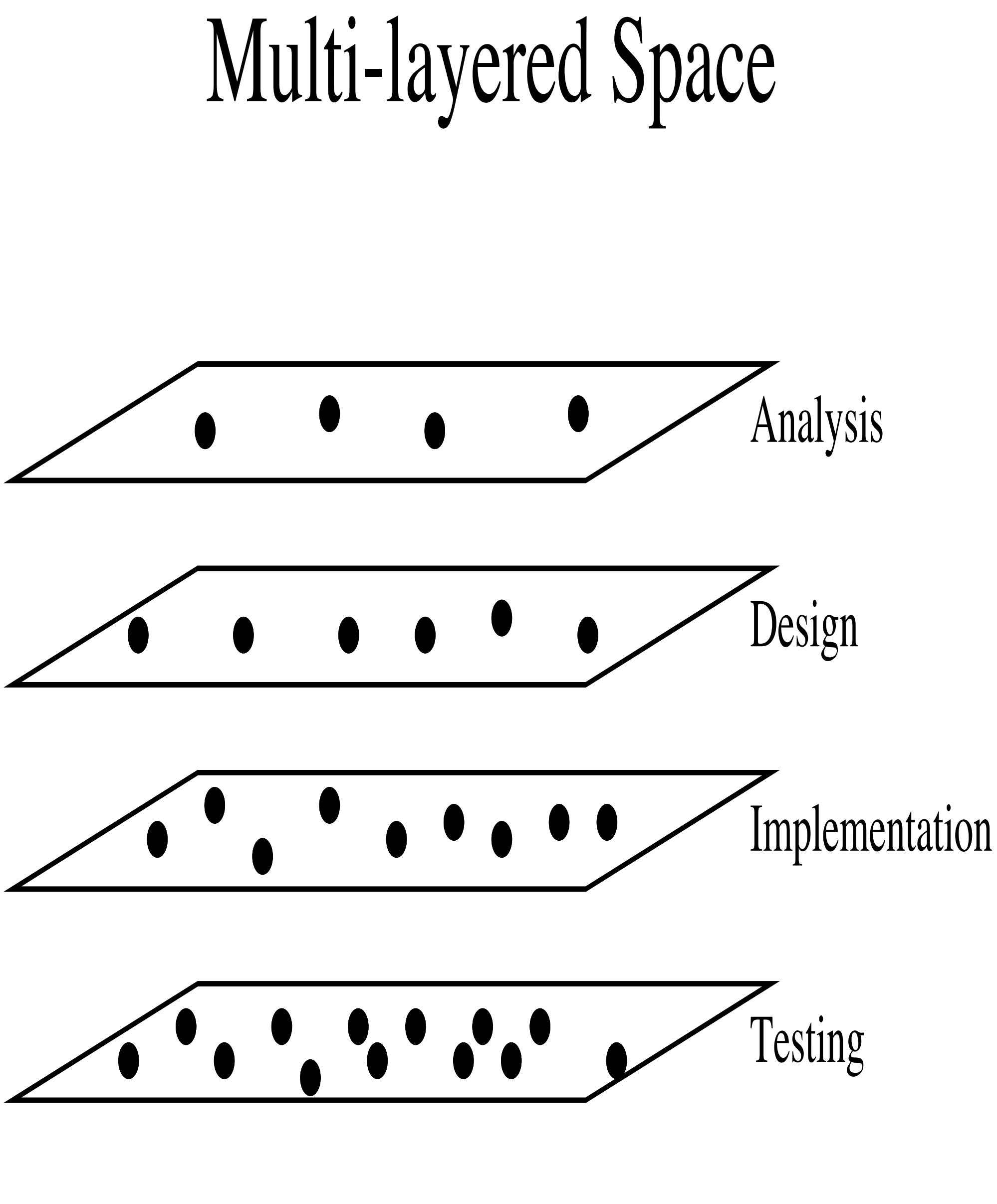


Figure 5.4: Multi-layered Space

Let

$$S^{t_0} \quad = \quad < A, D, I, T >$$

be the initial space, then S is static if

$$(\forall \; time \; t) \; S^t \quad = \quad S^{t_0}.$$

Definition 12 (Dynamic Space) *A dynamic space changes over time.*

In a dynamic space, steps may enter or leave the space at any time. Let

$$S^{t_0} \quad = \quad < A, D, I, T >$$

be the initial space, S is dynamic if

$$(\exists \; time \; t) \; S^t \quad \neq \quad S^{t_0}.$$

Definition 13 (Monotonic Property) *If in a dynamic space all newly introduced steps are consistent with existing steps, then the dynamic space is said to have the monotonic property.*

If in a dynamic space when additional steps are discovered, more work may be required to accomplish these steps but the newly discovered steps and their associated work are consistent additions to the system as defined by the already accomplished steps, then the space is monotonic. No part of the system has to be replaced or thrown out to accommodate the newly discovered steps.

Definition 14 (Non-monotonic Property) *If a dynamic space contains steps that are in conflict with each other, then the space is said to be non-monotonic.*

In a non-monotonic space, some steps may be in conflict with other steps. The choice of one of these steps will negate the other step even if the negated step is already considered part of the solution. The conflicts are primarily in the analysis plane between different requirements. However, conflicts in the design, implementation, and testing planes may also exist. Same plane conflicts may also occur.

Consider a non-monotonic conflict within the analysis plane. Let A_1 be a collection of analysis steps that are consistent with themselves. Let A_2 be a collection of analysis steps that are consistent with themselves but in conflict with the analysis steps found in A_1. To accomplish steps A_2 one would first have to mitigate conflicting steps A_1 and visa versa. Let A_3 be a collection of analysis steps that are consistent with themselves and consistent with the analysis steps in A_1 and the analysis step in A_2. There are then two consistent analysis options. Either analysis A is $< A_1, A_3 >$ or A is $< A_2, A_3 >$ but not both. Similar definitions apply for conflict within the design, implementation, and testing planes.

Consider a non-monotonic conflict that crosses many planes. Let D_1 be a collection of design steps, let I_1 be a collection of implementation steps, and let T_1 be a collection of testing steps that are all consistent with A_1. Let D_2 be a collection of design steps, let I_2 be a collection of implementation steps, and let T_2 be a collection of testing steps that are all consistent with A_2 but in conflict with A_1, D_1, I_1, or T_1. Let D_3 be a collection of design steps, let I_3 be a collection of implementation steps, and let T_3 be a collection of testing steps that are all consistent with A_3 and consistent with A_1, D_1, I_1, and T_1 but in conflict with A_2, D_2, I_2, and T_2. Then either one of the following holds but not both.

$$
\begin{aligned}
S &= \; < A_1, A_3, D_1, D_3, I_1, I_3, T_1, T_3 > \\
& \quad or \\
S &= \; < A_2, A_3, D_2, D_3, I_2, I_3, T_2, T_3 >
\end{aligned}
$$

Definition 15 (Problem Statement) *The problem statement is the highest level declarative goal of the system.*

Above the space, defined by the analysis, design, implementation, and testing planes, is the problem statement. The problem statement defines, at a very high level and abstraction, the declarative goal of the system. The problem statement is a requirement but because it is the parent step of all other steps, it is elevated above the analysis plane to emphasize importance. The problem statement may represent a compound step. In a dynamic space the problem statement may also change with time.

For clarity, a_1 will represent the problem statement in the remaining sections.

Definition 16 (System Acceptance Test) *The system acceptance test indicates the readiness of the system for product release.*

The system acceptance test is the final step in the testing plane. If the outcome of the system acceptance test is acceptable, then the system is ready for product release and general customer availability. The system acceptance test verifies that the requirements in the problem statement have been satisfied.

For clarity, t_{n_t} will represent the system acceptance test in the remaining sections.

Definition 17 (Environment) *The environment is the multi-layered finite space consisting of the analysis, design, implementation, and testing planes with two special steps: the initial problem statement and the system acceptance test.*

The environment is hierarchical with each node representing an atomic, compound, or complex step. The environment may be static or dynamic. See Figure 5.5 on page 77 for a visual representation of the environment.

To summarize the taxonomy, an environment contains the four disjoint finite planes of analysis, design, implementation, and testing. Each plane may contain many steps. Each plane contains at least one step. Each step may be atomic, compound, or complex. Steps may have sibling relationships. There is a special analysis step called the problem statement and a special testing step called the system acceptance test. See Figure 5.6 on page 78 for a visual representation of the taxonomy.

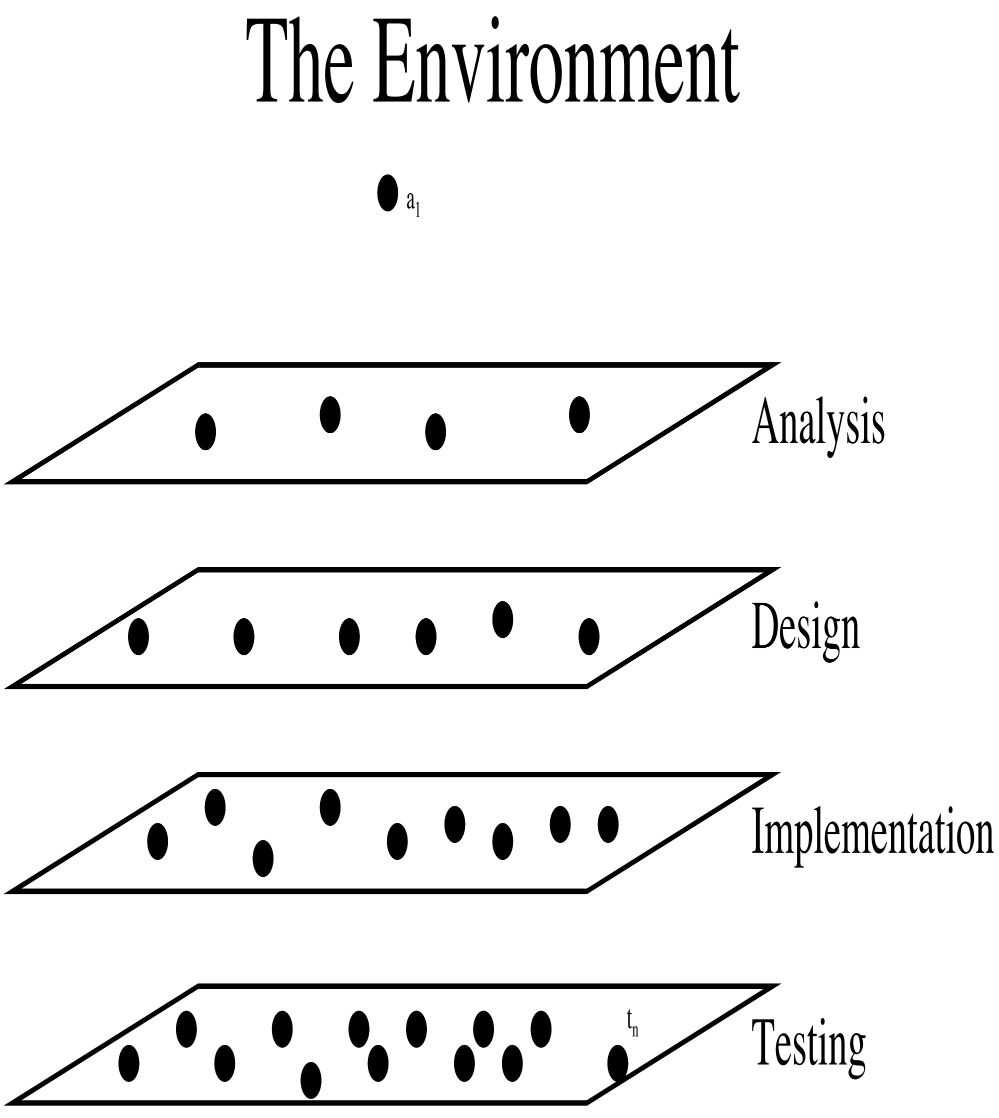


Figure 5.5: The Environment

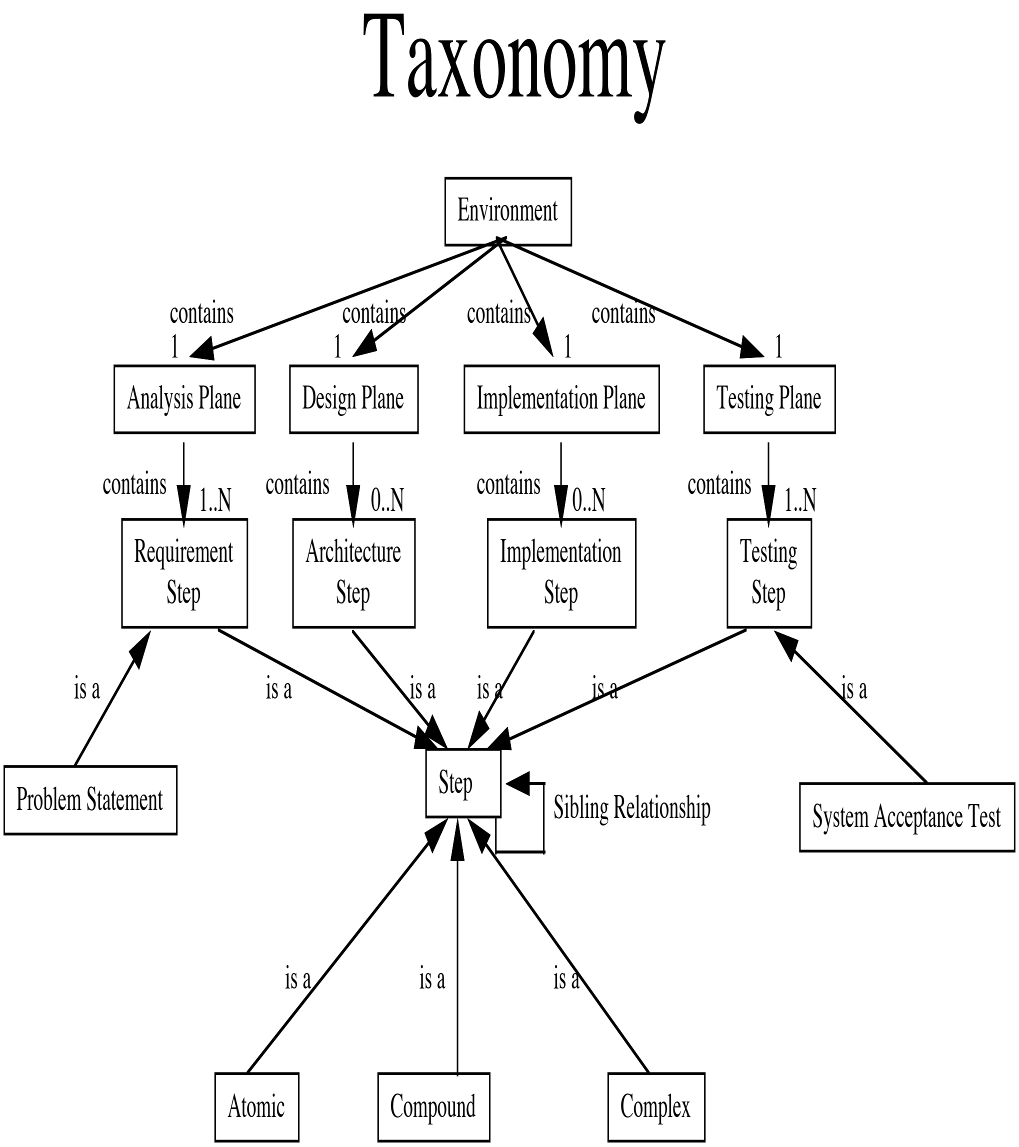


Figure 5.6: The Taxonomy

5.2.2 Towards the Definition of Methodology

Next we define the methodology, as soon discussed, as an algorithm that finds a solution in the given environment of the multi-layered finite space consisting of the analysis, design, implementation, and testing plane, starting with the root represented by the problem statement and ending with the goal represented by the system acceptance test. Three classes of methodologies, or algorithms, are presented: sequential, cyclical, and the WaterSluice.

Definition 18 (Solution) *A solution is a tree, or in a more general case, a directed acyclic graph, rooted at the problem statement and includes the system acceptance test that satisfies all of the goals in the problem statement.*

In a static environment, the solution may include all steps known in the environment. In a dynamic environment, the solution may be a subset. A solution does not contain any conflicting steps. The solution represents the final system, complete with analysis, design, implementation, and testing. A step visited but not used in the solution represents wasted effort. See Figure 5.7 on page 80 for a visual representation of a solution.

Definition 19 (Partial Solution) *A partial solution satisfies a consistent collection of goals in the problem statement.*

A partial solution does not satisfy all of the goals in the problem statement. If some of the goals in the problem statement are in conflict with each other, then only a partial solution exists. See Figure 5.8 on page 82 for a visual representation of a partial solution.

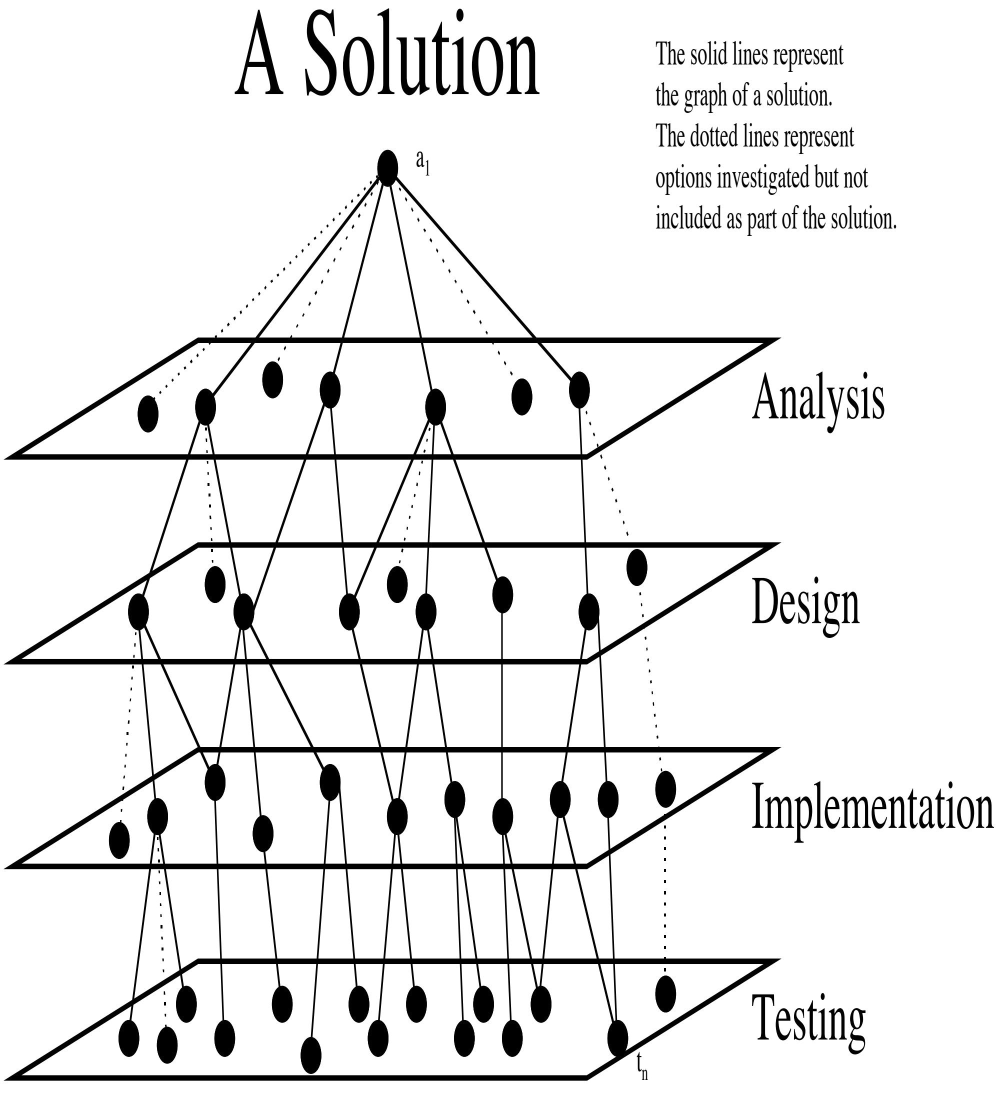


Figure 5.7: A Solution

Definition 20 (Optimal Solution) *An optimal solution satisfies all of the goals in the problem statement in an optimal fashion.*

If some of the goals in the problem statement are in conflict with each other, then no optimal solution exists.

Definition 21 (Feasible Solution) *A feasible solution satisfies all of the goals in the problem statement but not necessarily in an optimal fashion.*

There may be many feasible solutions to a given problem statement.

Assumption 1 *The space is finite and of size N.*

This thesis deals with software engineering of systems that can be built in a finite amount of time with finite resources using finite computers. Thus the assumption of a finite space is reasonable. Consider the alternative that the space is infinite. Software engineering systems simply can't be built that require an infinite number of steps.

This introduces the question of the size of this space. What is N?

Let N_1 be the number of steps in an optimal solution. Recall that an optimal solution satisfies all the goals in the problem statement.

A feasible solution satisfies all of the goals in the problem statement but not necessarily in an optimal fashion. A feasible solution has the same starting problem statement and the same ending system acceptance test as an optimal solution. Let N_2 be the upper bound on the number of steps over all feasible solution.

A partial solution satisfies some of the goals in the problem statement. A partial solution has the same ending system acceptance test as an optimal solution but satisfies only some of the goals of the problem statement. Let N_3 be the upper bound

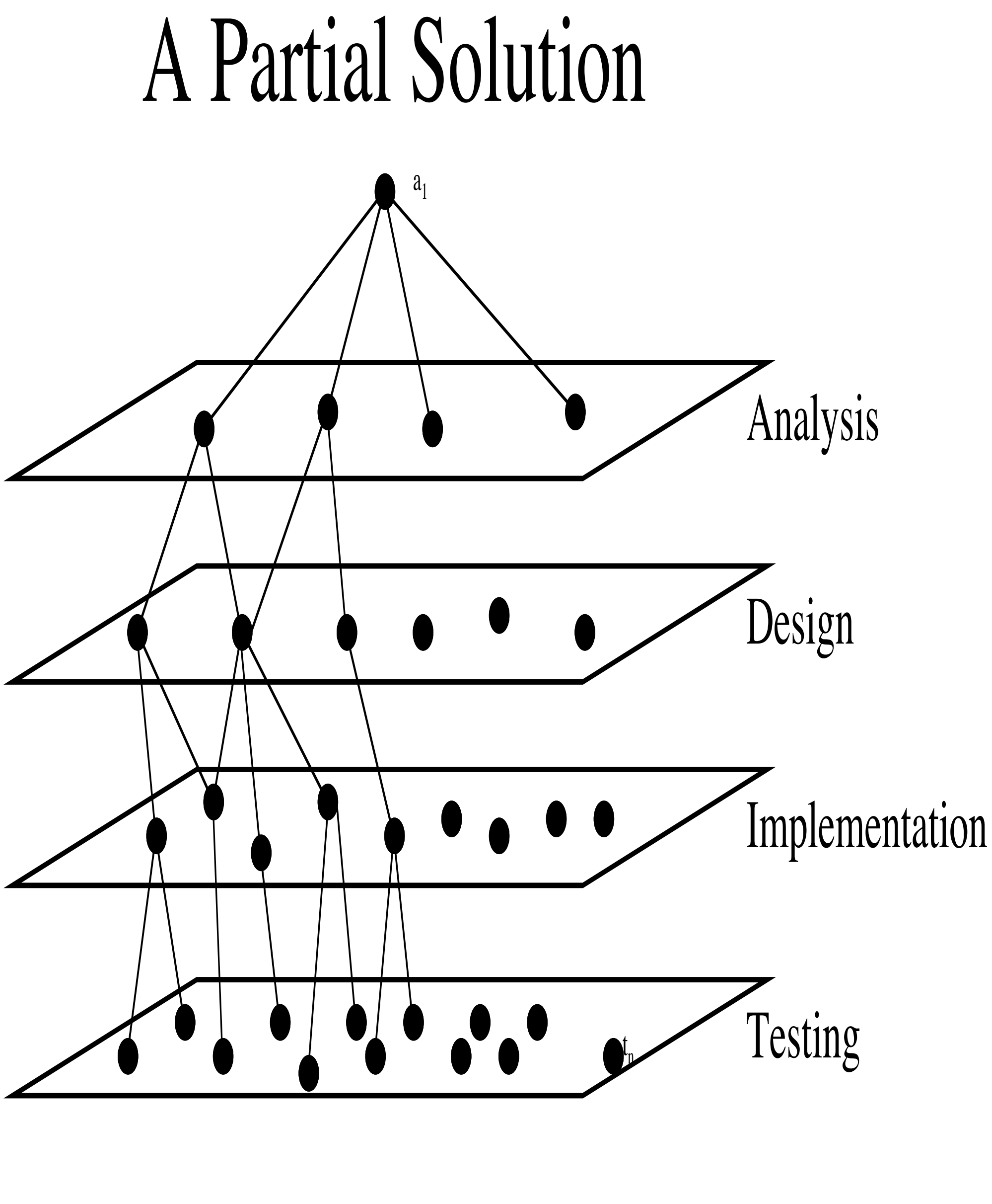


Figure 5.8: A Partial Solution

on the number of steps over all permutations and sub collections of the goals from the problem statement over all partial solutions.

Define the size of the space to be the upper bound of N_1, N_2, and N_3.

Definition 22 (Methodology) *A methodology is an algorithm that finds a feasible solution in the given environment of the multi-layered space consisting of the analysis, design, implementation, and testing plane, starting with the root represented by the problem statement and ending with the goal represented by the system acceptance test.*

The three main categories of software engineering methodologies under investigation are sequential, cyclical, and the WaterSluice.

Definition 23 (Sequential Methodology) *In a sequential software engineering methodology, all steps in the analysis plane are completed first, followed by all steps in the design plane, followed by all steps in the implementation plane, and then followed by all steps in the testing plane.*

A more detailed description can be found in Section 3.1. Also see Figure 3.1 on page 29 for a graphical representation of a sequential methodology.

Definition 24 (Cyclical Methodology) *A cyclical software engineering methodology cycles through each phase a few steps at a time until a feasible solution is established.*

Simply, a cyclical software engineering methodology iterates over the processes of think a little, plan a little, implement a little, then test a little. Finer and finer details are generated as the cyclical software engineering methodology progresses. Finally, after several iterations, the system is completed.

A more detailed description can be found in Section 3.2. Also see Figure 3.2 on page 33 for a graphical representation of a cyclical methodology.

Definition 25 (WaterSluice) *The WaterSluice combines the steady progression of the sequential software engineering methodology with the iterative nature of the cyclical software engineering methodology while adding priority. Non-monotonic conflicts are handled by change order control.*

The WaterSluice software engineering methodology separates the important aspects from the less important and concentrates on solving them first. As the process continues, finer and finer details are refined until the product is released.

A more detailed description can be found in Section 3.3.1. Also see Figure 3.4 on page 37 for a graphical representation of the WaterSluice methodology.

5.2.3 Towards the Definition of Performance

Finally, we define performance, as soon discussed, as the number of steps needed by a methodology, an algorithm, to find a solution.

Definition 26 (Complete) *If a solution exists, and the software engineering methodology can find a solution for every environment, then the software engineering methodology is said to be complete.*

Definition 27 (Performance) *If a solution exists, the performance of the software engineering methodology is defined as the number of steps needed to find a feasible solution.*

A step may be atomic, compound, or complex and be in any of the analysis, design, implementation, or testing planes. The performance measurement of counting steps assumes that on the average each step the same amount of effort. Since the performance is an order-of-magnitude measurement, knowledge of the detailed effort of each step is not necessary. This assumption is similar to the assumption that all instructions take equal time when doing algorithm performance analysis. Of course, a more detailed performance measurement could be defined where different weights can be used for different steps.

To achieve the best case performance, an environment is created that will allow the methodology to find a solution in the least number of steps. In a worst case performance, an environment is created that will prevent the methodology from finding a solution until the entire environment is visited. In an average case performance, environments are created for the methodology that requires a typical number of steps to find a feasible solution.

5.3 Supporting Theorems

A family of theorems and corollaries are now proven. It will be soon shown that all three categories of software engineering methodologies are complete for static environments. Only cyclical and WaterSluice are complete for dynamic environments while only WaterSluice is complete for non-monotonic environments. The best case performance of the sequential methodology is O(N). The best case performance of the cyclical and the WaterSluice methodology is O(1). The worst case performance of all three categories of methodologies are the same. On average the sequential methodology will find a solution in O(N), the cyclical methodology will find a solution in O(N), [1] and the WaterSluice will find a solution in an order-of-magnitude less than N.

5.3.1 Sequential Software Engineering Methodology

In this section, a family of theorems are presented that pertain to the sequential software engineering methodology. The sequential software engineering methodology will find solutions in static environments but not in dynamic environments. If a solution exists, and the sequential software engineering methodology finds the solution, the best case performance is $O(N)$, the worst case performance is $O(N)$, while the average case performance is $O(N)$ where N is the total number of steps in the environment.

Theorem 2 (Sequential: Static Complete) *A sequential software engineering methodology is static complete.*

[1] A more accurate average performance measurement for a cyclical software engineering methodology is N/2.

Let $a_1, a_2, a_3, \ldots, a_{n_a}$ be the n_a requirements leading to a possible analysis A with a_1 the initial problem statement. Let $d_1, d_2, d_3, \ldots, d_{n_d}$ be the n_d architecture elements leading to a possible design D. Let $i_1, i_2, i_3, \ldots, i_{n_i}$ be the n_i implementation components leading to a possible implementation I. Let $t_1, t_2, t_3, \ldots, t_{n_t}$ be the n_t testing suites leading to a possible testing T with t_{n_t} being the final system acceptance test. Define a static environment that consists of the four planes of analysis, design, implementation, and testing with each lower plane being a refinement of the higher planes. Define the solution as the tree, or in a more general sense, the directed acyclic graph, of all steps from all four planes used in the final system. This solution can be represented as a sequence of steps $< A, D, I, T >$.

See Figure 5.5 on page 77 for a visual representation of the static environment. See Figure 5.9, Figure 5.10, and Figure 5.11, on pages 88 through 90 for a visual guide of the reasoning. The numbers associated with the steps represent the order visited by the methodology.

Proof 2.1 *A sequential software engineering methodology leads to a sequence of steps.*

step 1*:*

$$S^1 = < a_1 >$$

step 2*:*

$$S^2 = < a_1, a_2 >$$

$\vdots$

step n_a*:*

$$\begin{aligned} S^{n_a} &= < a_1, a_2, \ldots, a_{n_a} > \\ &= < A > \end{aligned}$$

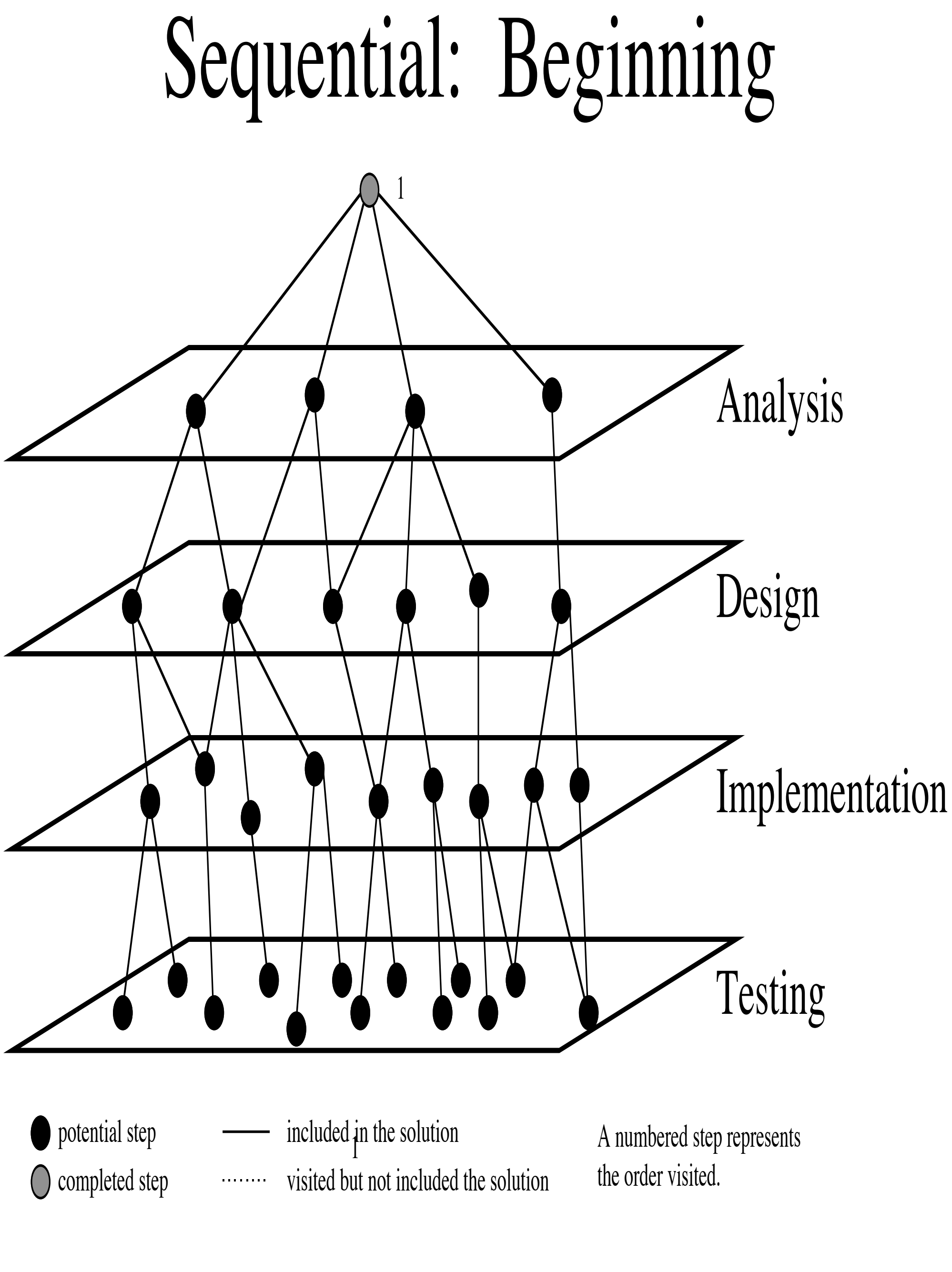


Figure 5.9: Sequential: Beginning

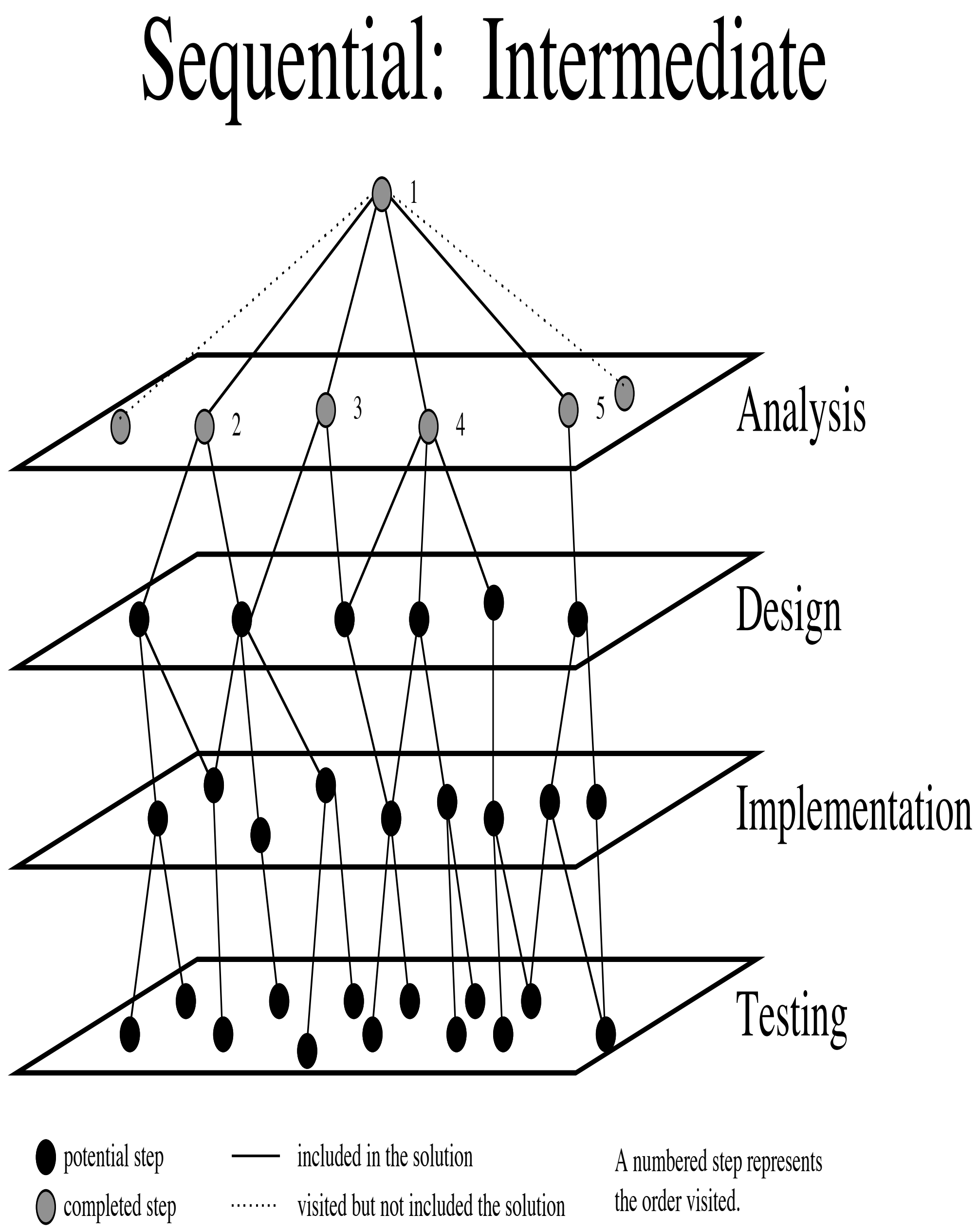


Figure 5.10: Sequential: Intermediate

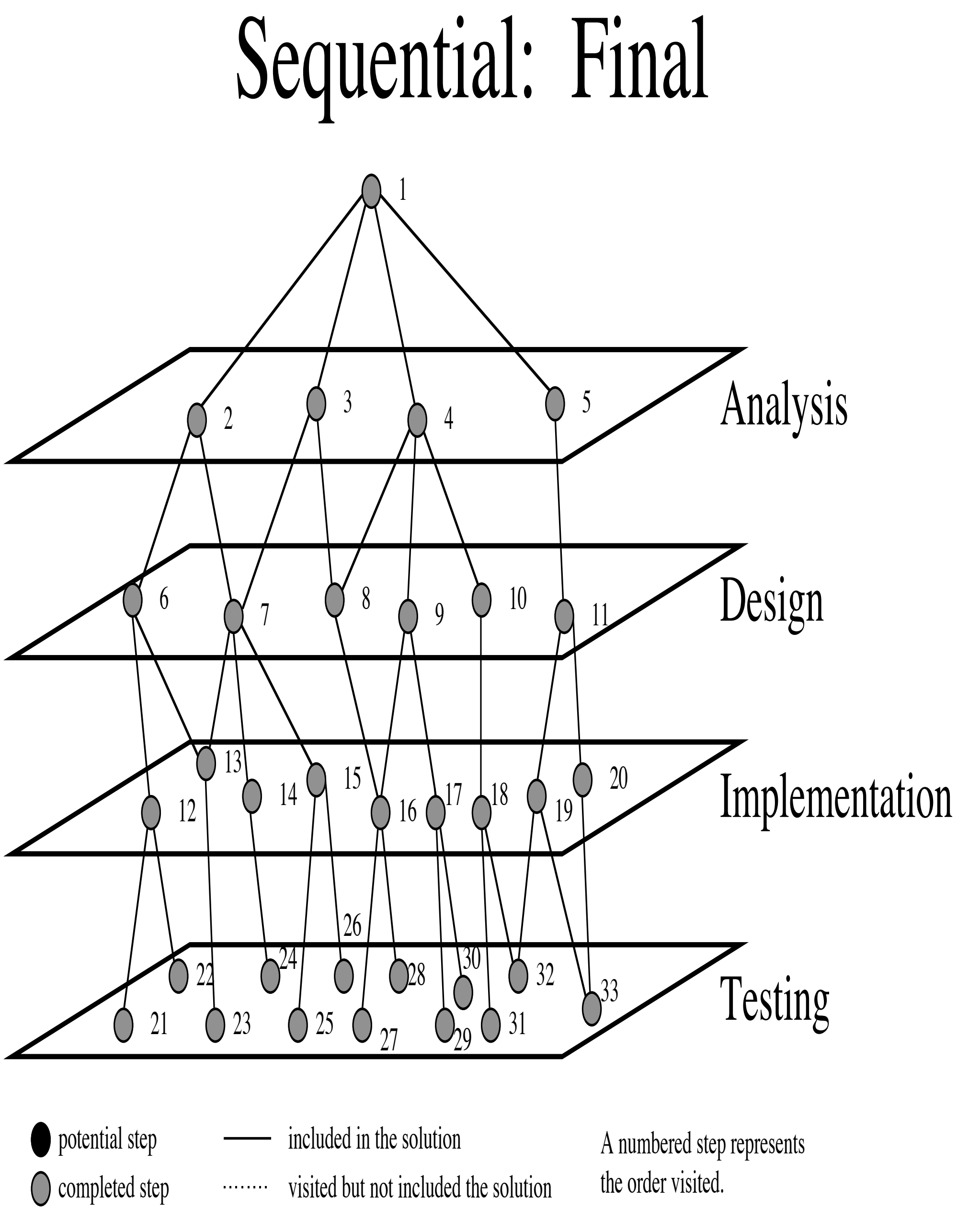


Figure 5.11: Sequential: Final

$\vdots$

step $n_a + n_d$:

$$\begin{aligned} S^{n_a+n_d} &= < A, d_1, d_2, \ldots, d_{n_d} > \\ &= < A, D > \end{aligned}$$

$\vdots$

step $n_a + n_d + n_i$:

$$\begin{aligned} S^{n_a+n_d+n_i} &= < A, D, i_1, i_2, \ldots, i_{n_i} > \\ &= < A, D, I > \end{aligned}$$

$\vdots$

step $n_a + n_d + n_i + n_t$:

$$\begin{aligned} S^{n_a+n_d+n_i+n_t} &= < A, D, I, t_1, t_2, \ldots, t_{n_t} \\ &= < A, D, I, T > \end{aligned}$$

The sequence represented by $< A, D, I, T >$ is the solution.

These steps find a solution where steps at the higher levels are exhausted first before going into lower levels. If the environment is static, then all steps in the environment are known before the sequential methodology begins. The sequential methodology first discovers all steps in the analysis plane, followed by all steps in the design plane, followed by all steps in the implementation plane, and then followed by all steps in the testing plane. This can be accomplished because the environment is static and finite. Eventually in the static environment the solution is discovered. Thus, the sequential software engineering methodology is static complete.

Corollary 2.1 (Sequential: Best Case) *The best case performance of the sequential software engineering methodology is O(N).*

Proof 2.1.1 *Let the environment consist of n_a analysis steps, n_d design steps, n_i implementation steps, and n_t testing steps. Let the system acceptance test be the very fist step in the testing plane t_1. The number of steps to find a solution is $n_a + n_d + n_i + 1$ which is O(N).*

Corollary 2.2 (Sequential: Worst Case) *The worst case performance of the sequential software engineering methodology is O(N) where N is the size of the space.*

Proof 2.2.1 *Let the environment consist of n_a analysis steps, n_d design steps, n_i implementation steps, and n_t testing steps. Let the system acceptance test be the very last step t_{n_t}. The number of steps to find a solution is $n_a + n_d + n_i + n_t$ which is O(N).*

Corollary 2.3 (Sequential: Average Case) *The average case performance of the sequential software engineering methodology is O(N) where N is the size of the space.*

Proof 2.3.1 *Let the environment consist of n_a analysis steps, n_d design steps, n_i implementation steps, and n_t testing steps. Let the system acceptance test be the $(n_t)/2$ step in the testing plane $t_{(n_t)/2}$. The number of steps to find a solution is $n_a + n_d + n_i + (n_t)/2$ which is O(N).*

Corollary 2.4 (Sequential: Dynamic Incomplete) *The sequential software engineering methodology may not find a solution in a dynamic environment.*

In a sequential software engineering methodology, all of the high level steps are exercised before proceeding to the next plane of the space. If a new step is introduced once the sequential software engineering methodology has finished with this plane of the environment, the new step will not be part of the solution. The sequential methodology will no longer be able to find the solution.

Let the initial environment consist of n_a analysis steps, n_d design steps, n_i implementation steps, and n_t testing steps.

Proof 2.4.1 *At step $n_a + 1$ the sequential software engineering methodology has accomplished the steps $< a_1, a_2, \ldots, a_{n_a}, d_1 >$. At this time introduce a new requirement a_{n_a+1}. The sequential software engineering methodology has already finished with the analysis plane and will not discover this new requirement. Thus, a sequential software engineering methodology is dynamic incomplete.*

Corollary 2.5 (Sequential: Non-monotonic Incomplete) *The sequential software engineering methodology may not find a solution in a non-monotonic environment.*

Proof 2.5.1 *The sequential software engineering methodology is incomplete in dynamic environment and thus incomplete to dynamic environments with the non-monotonic property.*

5.3.2 Cyclical Software Engineering Methodology

In this section, a family of theorems are presented that pertain to the cyclical software engineering methodology. The cyclical software engineering methodology will find solutions in both static and dynamic environments but not in dynamic environments that have the non-monotonic property. If a solution exists, and the cyclical software engineering methodology finds the solution, the best case performance is $O(1)$, the worst case performance is $O(N)$, while the average case performance is $O(N)$ [2] where N is the total number of steps in the environment.

Theorem 3 (Cyclical: Static Complete) *A cyclical software engineering methodology is static complete.*

Let $a_1, a_2, a_3, \ldots, a_{n_a}$ be the n_a steps leading to the possible analysis A with a_1 being the initial problem statement. Let $d_1, d_2, \ldots, d_{n_d}$ be the n_d steps leading to the possible design D. Let $i_1, i_2, \ldots, i_{n_i}$ be the n_i steps leading to the possible implementation I. Let $t_1, t_2, \ldots, t_{n_t}$ be the n_t steps leading to the possible testing T with t_{n_t} being the final system acceptance test. Though this static environment includes the same steps as in the other sections, these steps are not necessarily taken in the same sequence as those established in the other sections but are the sequence taken by a cyclical methodology. A cyclical methodology has a simple selection process which defines this order. Define a static environment that consists of the four planes of analysis, design, implementation, and testing with each lower plane being a refinement of the higher planes. Define the solution as the tree, or in a more general sense, the directed acyclic graph, of all steps from all four planes used in the final system. This solution can be represented as a sequence of steps. See Figure 5.5 on page 77 for

---

[2] To be more precise, the exact performance is $N/2$.

a visual representation of the environment. See Figure 5.12 on page 96 for a visual guide of the reasoning. The numbers associated with the steps represent the order visited by the methodology.

Proof 3.1 *A cyclic methodology leads to this sequence of steps.*

step 1*: Initial problem statement.*

$$S^1 = < a_1 >$$

step 2*:*

$$S^2 = < a_1, a_2 >$$

step 3*:*

$$S^3 = < a_1, a_2, d_1 >$$

step 4*:*

$$S^4 = < a_1, a_2, d_1, i_1 >$$

step 5*:*

$$S^5 = < a_1, a_2, d_1, i_1, t_1 >$$

$\vdots$

The last step:

$$\begin{aligned} S^{n_a+n_d+n_i+n_t} &= < a_1, a_2, d_1, i_1, t_1, \ldots, a_{n_a}, d_{n_d}, i_{n_i}, t_{n_t} > \\ &= < A, D, I, T > \end{aligned}$$

The last step is the system acceptance test. Thus a cyclical methodology is static complete.

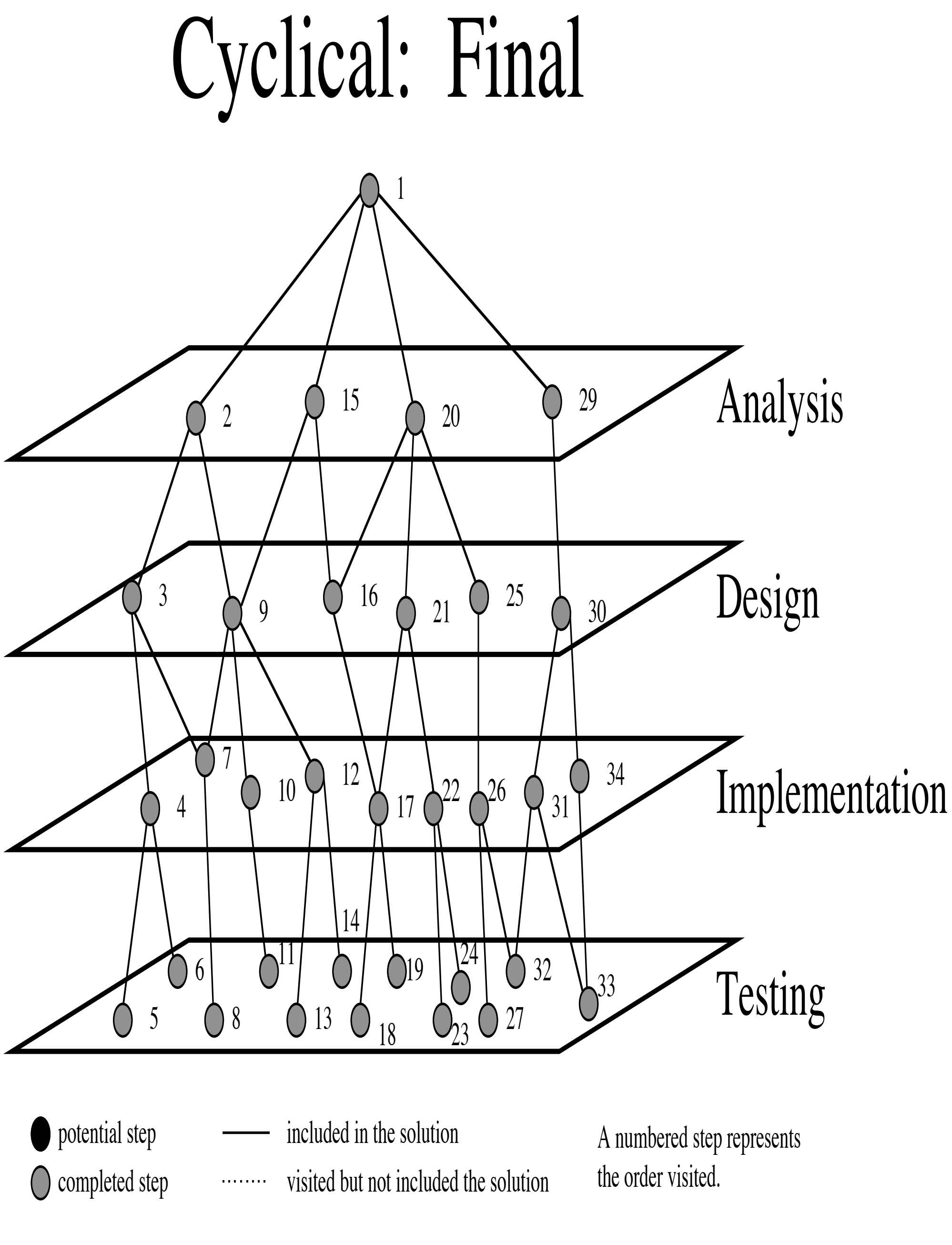


Figure 5.12: Cyclical: Final

These steps are generated in an iterative fashion and eventually exhaust the static environment.

If the environment is static, then all steps in the environment are known before the cyclical methodology begins. The cyclical methodology discovers steps in an iterative fashion in the analysis plane, the design plane, the implementation plane, and the testing plane. This iterative discovery of steps from all four planes continues until the environment was exhausted.

Corollary 3.1 (Cyclical: Best Case) *The best case performance of the cyclical software engineering methodology is O(1).*

Proof 3.1.1 *Let the environment consist of n_a analysis steps, n_d design steps, n_i implementation steps, and n_t testing steps. Let the correct analysis step be the very first step a_1. Let the correct design step be the very first step d_1. Let the correct implementation step be the very first step i_1. Let the system acceptance test be the very first step t_1. To satisfy the problem statement would require one analysis, one design, one implementation, and one testing step. In this environment, the cyclical software engineering methodology will require only four steps. Thus the performance is O(1).*

Corollary 3.2 (Cyclical: Worst Case) *The worst case performance of the cyclical software engineering methodology is O(N) where N is the size of the environment.*

Proof 3.2.1 *Let the environment consist of n_a analysis steps, n_d design steps, n_i implementation steps, and n_t testing steps. Let the system acceptance test be the very last step t_{n_t}. The number of steps to find a solution is $n_a + n_d + n_i + n_t$ which is O(N).*

Corollary 3.3 (Cyclical: Average Case) *The average case performance of the cyclical software engineering methodology is O(N)* [3] *where N is the size of the space.*

Proof 3.3.1 *Let the environment consist of* n_a *analysis steps,* n_d *design steps,* n_i *implementation steps, and* n_t *testing steps. On average, the system acceptance would be in the middle of the testing plane* $t_{n_t/2}$. *On average, the cyclical software engineering methodology would only have to discover half of the analysis, design, implementation, and testing steps. That is to say, half the time the cyclical software engineering methodology discovers less than half of the space to find a solution and half the time the cyclical software engineering methodology discovers more than half of the space to find a solution. The number of steps to find a solution is* $(n_a)/2 + (n_d)/2 + (n_i)/2 + (n_t)/2$ *which is* $N/2$ *steps. Thus, the performance is O(N).* [4]

Corollary 3.4 (Cyclical: Dynamic Complete) *A cyclical software engineering methodology is dynamic complete.*

Proof 3.4.1 *A cyclical methodology allows for the introduction of new information at every cycle and the removal of information that is no longer needed. Thus, for a dynamic environment, a cyclical software engineering methodology will eventually find the solution.*

Corollary 3.5 (Cyclical: Non-monotonic Incomplete) *A cyclical software engineering methodology is non-monotonic incomplete.*

[3] *To be more precise, the exact performance is* $N/2$.
[4] *To be more precise, the exact performance is* $N/2$.

Proof 3.5.1 *The cyclical software engineering methodology has no mechanism to manage a non-monotonic step. Consider a dynamic environment with two highly conflicting requirements and their associated design, implementation, and testing steps. The cyclical software engineering methodology would pick one requirement and build the associated system. The cyclical software engineering methodology, because there is no mechanism to manage a non-monotonic step, then picks the second conflicting requirement. The first solution is negated in order to build a second system. The cyclical software engineering methodology oscillates between these two systems with no convergence to a common solution. The final system acceptance test fails because one of the conflicting requirements can never be accomplished. Thus, the cyclical software engineering methodology is incomplete for a non-monotonic environment.*

An Example of Cyclical Non-feasible Solution

Many applications are partitioned into the three major components: user interface, application logic, and a database. The visible component of the application to a customer is the user interface. The other two components have no visibility to the customer and are hidden. In many cases the customer's mental model of the application is totally user-interface centric. To the customer, the application is the user interface.

Tools exist today to easily craft user interfaces. These tools ignore the building of the application logic and databases. The application logic and databases are crafted using more traditional techniques accomplished by trained individuals. An inexperienced programmer is easily convinced that the application is totally done when only the user interface is completed.

The problem comes when an inexperienced team is using one of these user interface tools along with the customer. Both the inexperienced team and the customer have highly slanted, user interface centric, mental models of the application. The team and the customer could spend hours working on very minuscule details of the user interface design by quickly iterating between different user interface designs. Yet the application logic and the supporting database may get no attention. Both the customer and the inexperienced team come to the false impression that they are near completion of application when the user interface is completed. They are stuck on user interface design details without getting a global picture of the entire application that includes application logic and a database.

Much later in this process the discovery could be made that the application logic and supporting database may be impossible to implement given the current user interface. Hours of user interface design could have been avoided if a more global view of the application had been introduced earlier.

A cyclical methodology refines some details early in the process without having a global view of the entire search space. In this case, the user interface was explored in detail without much attention to the application logic or the database design. This gives a false impression of near completion and progress even though the solution is stuck at a non-feasible position that excludes major components of the application. The important requirements of application logic and the database were missed until late in a cyclical methodology.

5.3.3 WaterSluice Software Engineering Methodology

In this section, a family of theorems are presented that pertain to the WaterSluice software engineering methodology. The WaterSluice software engineering methodology will find solutions in static environments and dynamic environments including dynamic environments that have the non-monotonic property. If a solution exists and the WaterSluice software engineering methodology finds the solution, the best case performance is $O(1)$, the worst case performance is $O(N)$, while the average case performance is an order-of-magnitude less than N where N is the total number of steps in the environment.

Theorem 4 (WaterSluice: Static Complete) *The WaterSluice software engineering methodology is static complete.*

Let $a_1, a_2, a_3, \ldots, a_{n_a}$ be the n_a steps leading to the possible analysis A with a_1 being the initial problem statement. Let $d_1, d_2, \ldots, d_{n_d}$ be the n_d steps leading to the possible design D. Let $i_1, i_2, \ldots, i_{n_i}$ be the n_i steps leading to the possible implementation I. Let $t_1, t_2, \ldots, t_{n_t}$ be the n_t steps leading to the possible testing T with t_{n_4} being the final system acceptance test. Define an environment that consists of the four planes of analysis, design, implementation, and testing with each lower plane being a refinement of the higher planes. In this environment priorities are represented as different in the size of a step. A larger circle represents a higher priority than a smaller circle. Define the solution as the tree, or in a more general sense, the directed acyclic graph, of all steps from all four planes used in the final system. Define a priority queue of possible next steps PQ. Initially the priority queue PQ contains the initial step a_1 represented by the sequence S^1. Define a function, *next*, over all possible steps, which generates all possible next steps given the current solution sequence, evaluates the total value, and pushes the ordered result onto the priority queue PQ. Only high priority steps are keep in the queue PQ. See Figure 5.13 on page 103 for a visual representation of the space. Here a larger circle represents a higher priority step. See figures 5.13, 5.15, 5.16, 5.17, and 5.18 on pages 103 through 108 for a visual guide of the reasoning. The numbers associated with the steps represent the order visited by the methodology.

Proof 4.1 *The WaterSluice methodology leads to this sequence of steps.*

step 1: Initial set up.

$$PQ = a_1$$

step 2: Remove a_1 from the priority queue PQ. Expand all alternative steps near a_1 using the function next and push them onto the priority queue PQ.

$$\begin{aligned} PQ &= a_2, a_3 \\ S^1 &= a_1 \end{aligned}$$

At this stage there are two alternative next steps. Pick the next step with the highest priority. For clarity, assume this step is a_2.

step 3:

$$\begin{aligned} PQ &= a_3, a_4 \\ S^2 &= a_1, a_2 \end{aligned}$$

⋮

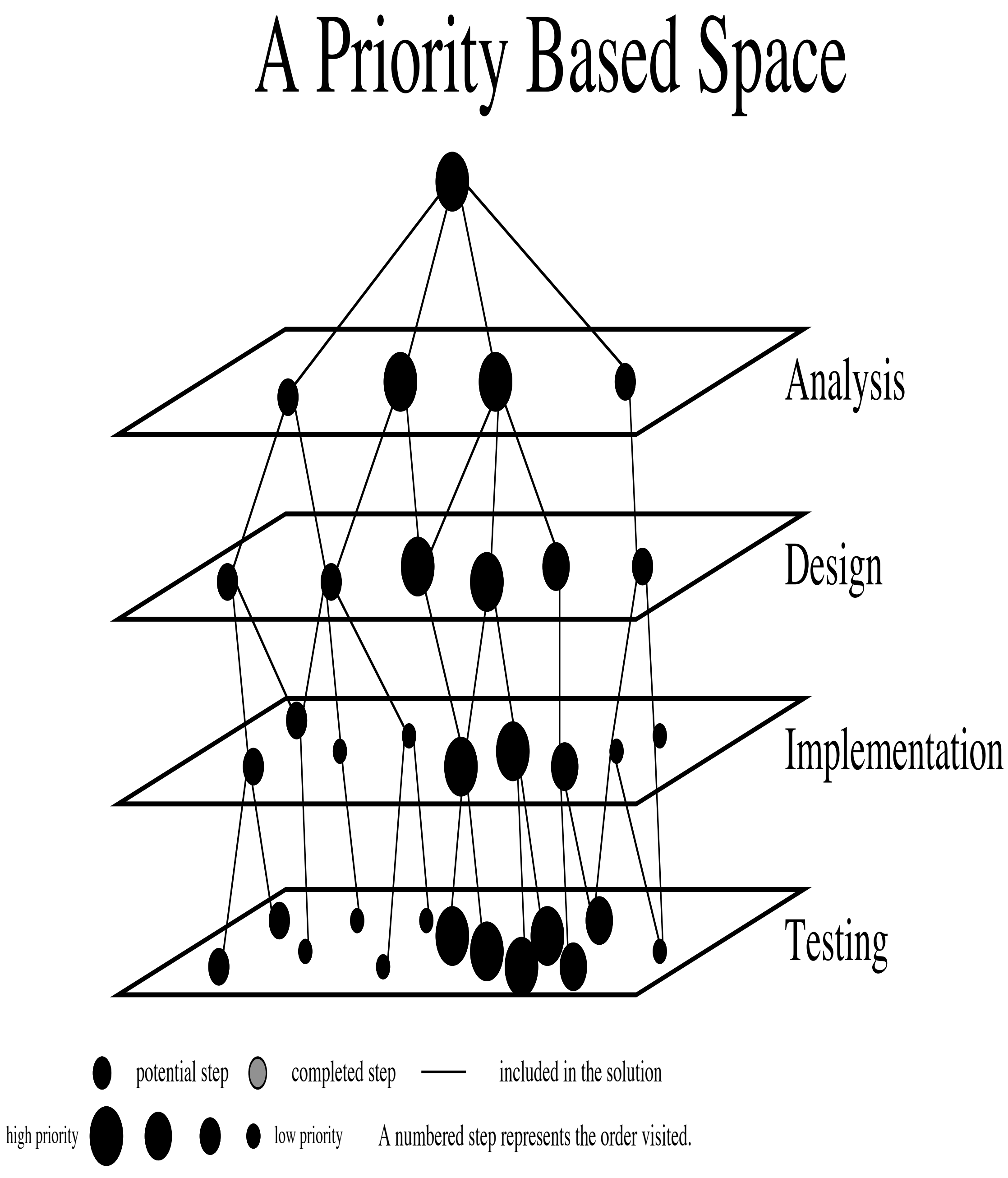


Figure 5.13: Priority Based Space

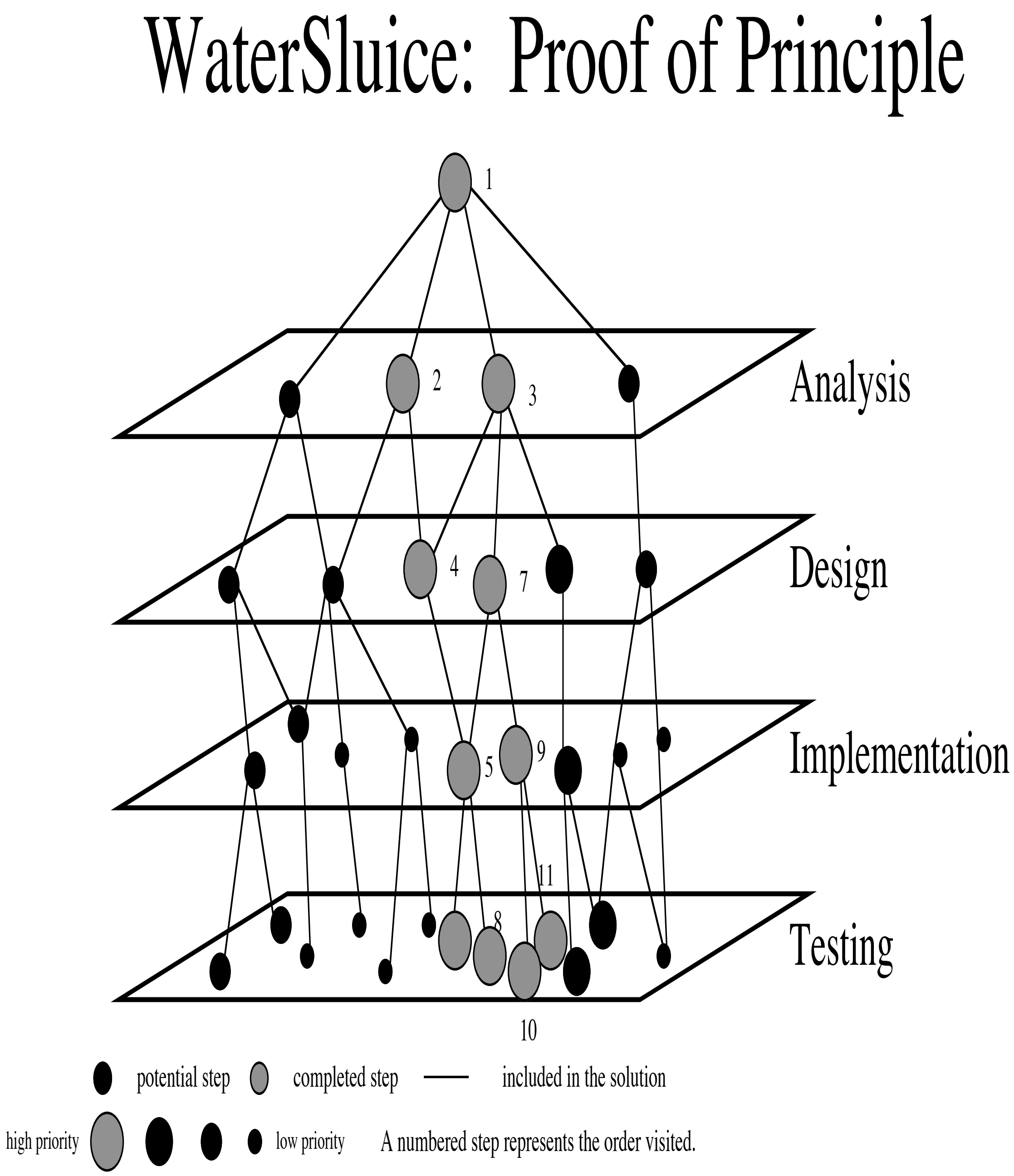


Figure 5.14: WaterSluice: Proof of Principle

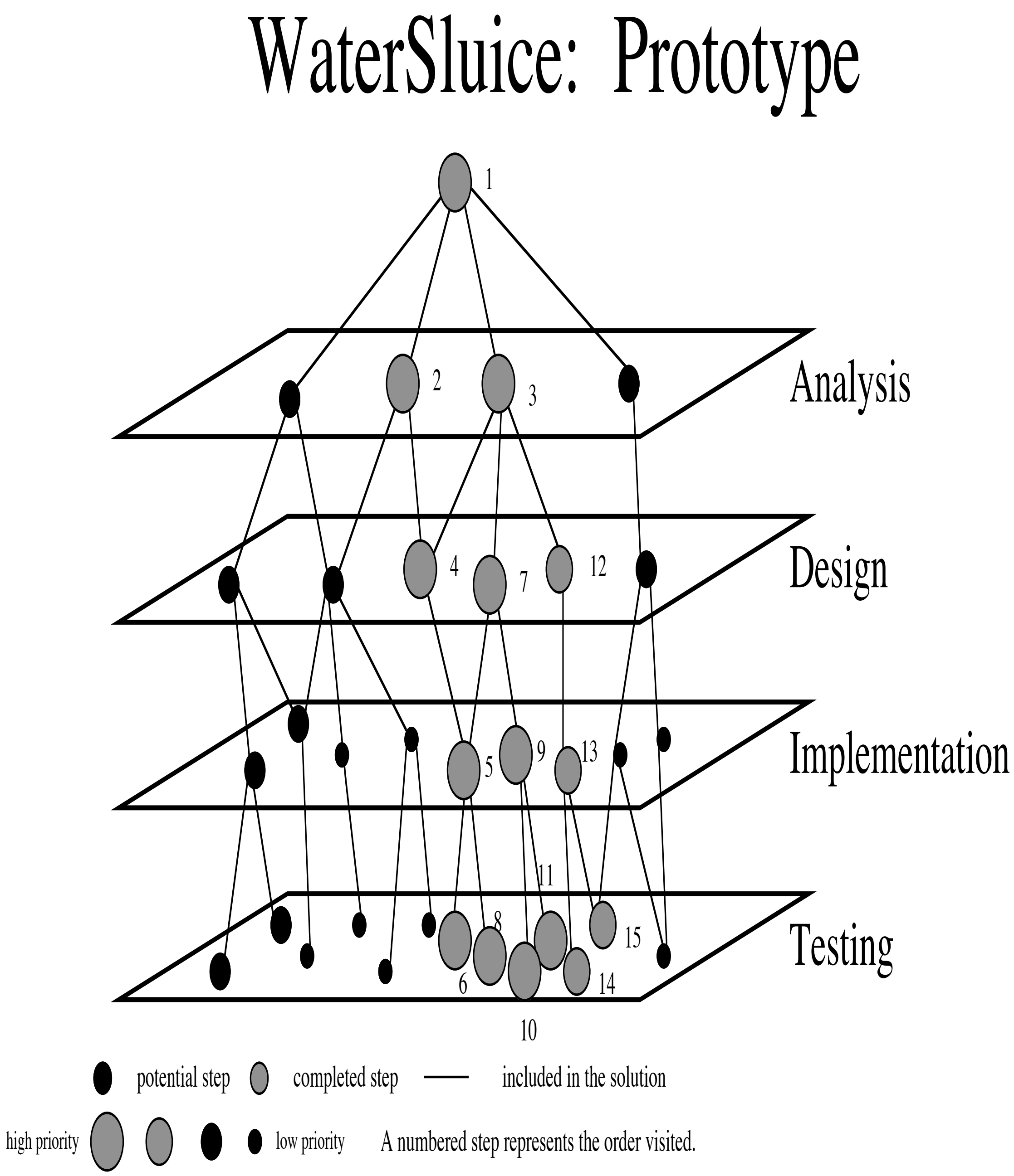


Figure 5.15: WaterSluice: Prototype

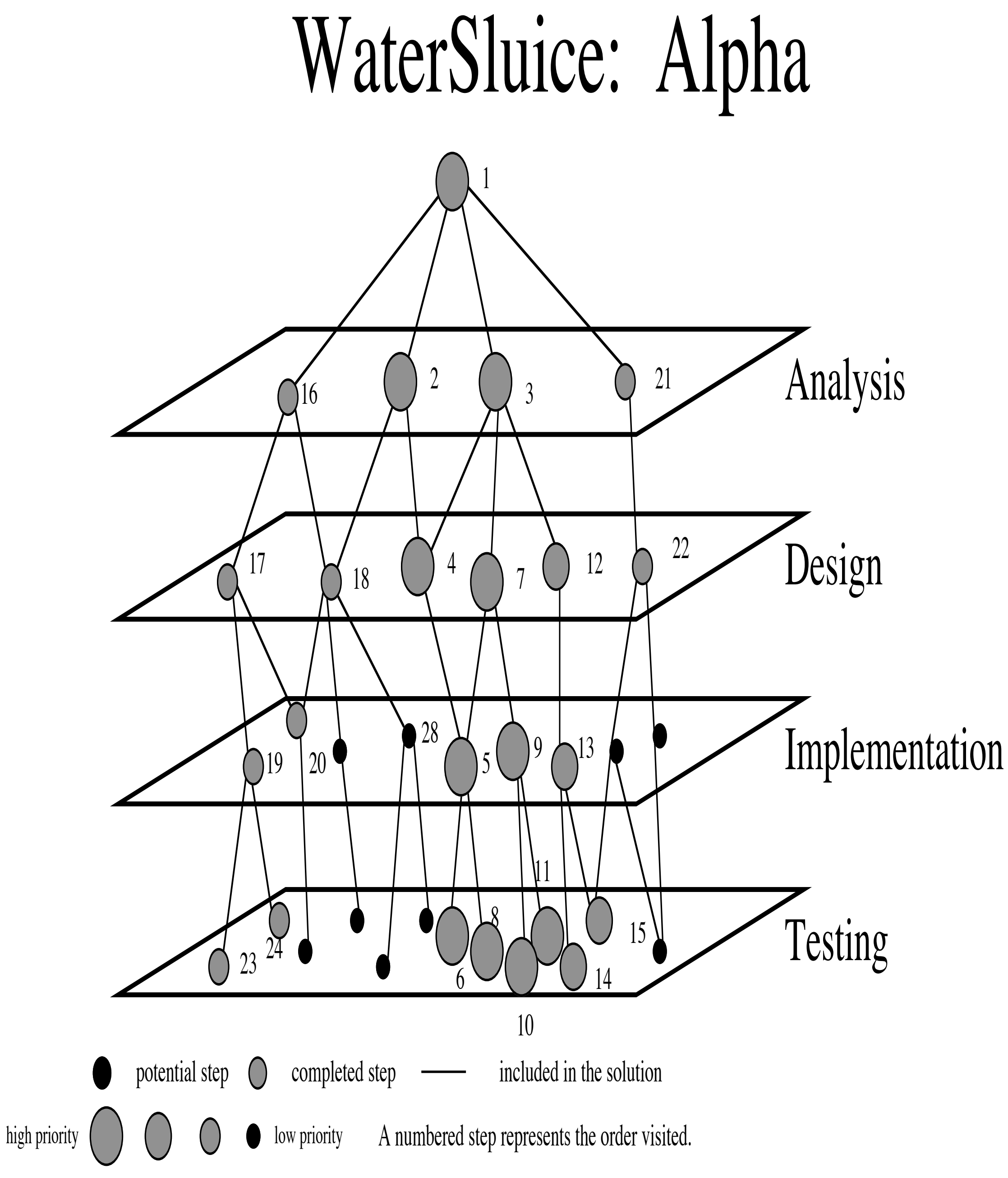


Figure 5.16: WaterSluice: Alpha

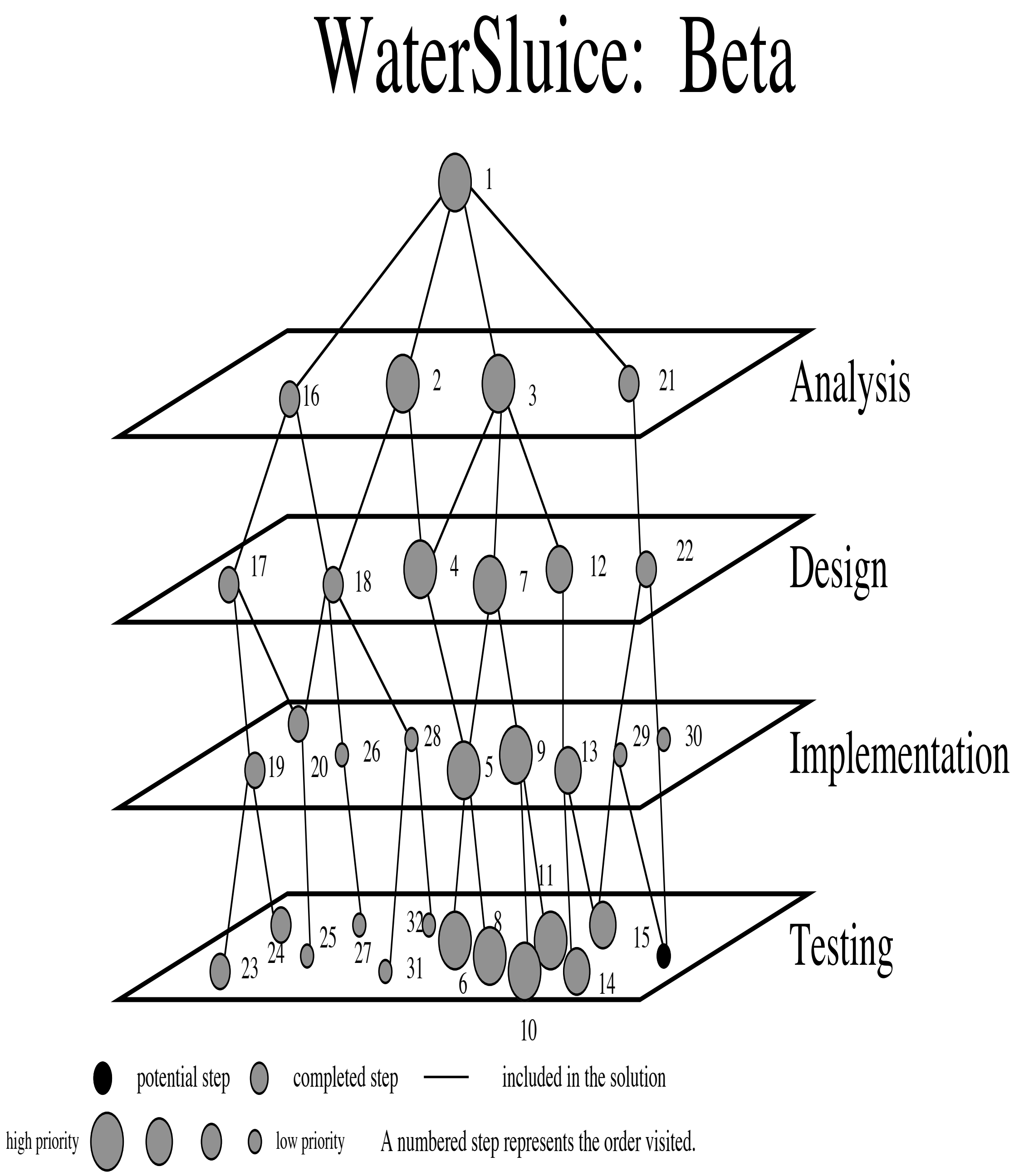


Figure 5.17: WaterSluice: Beta

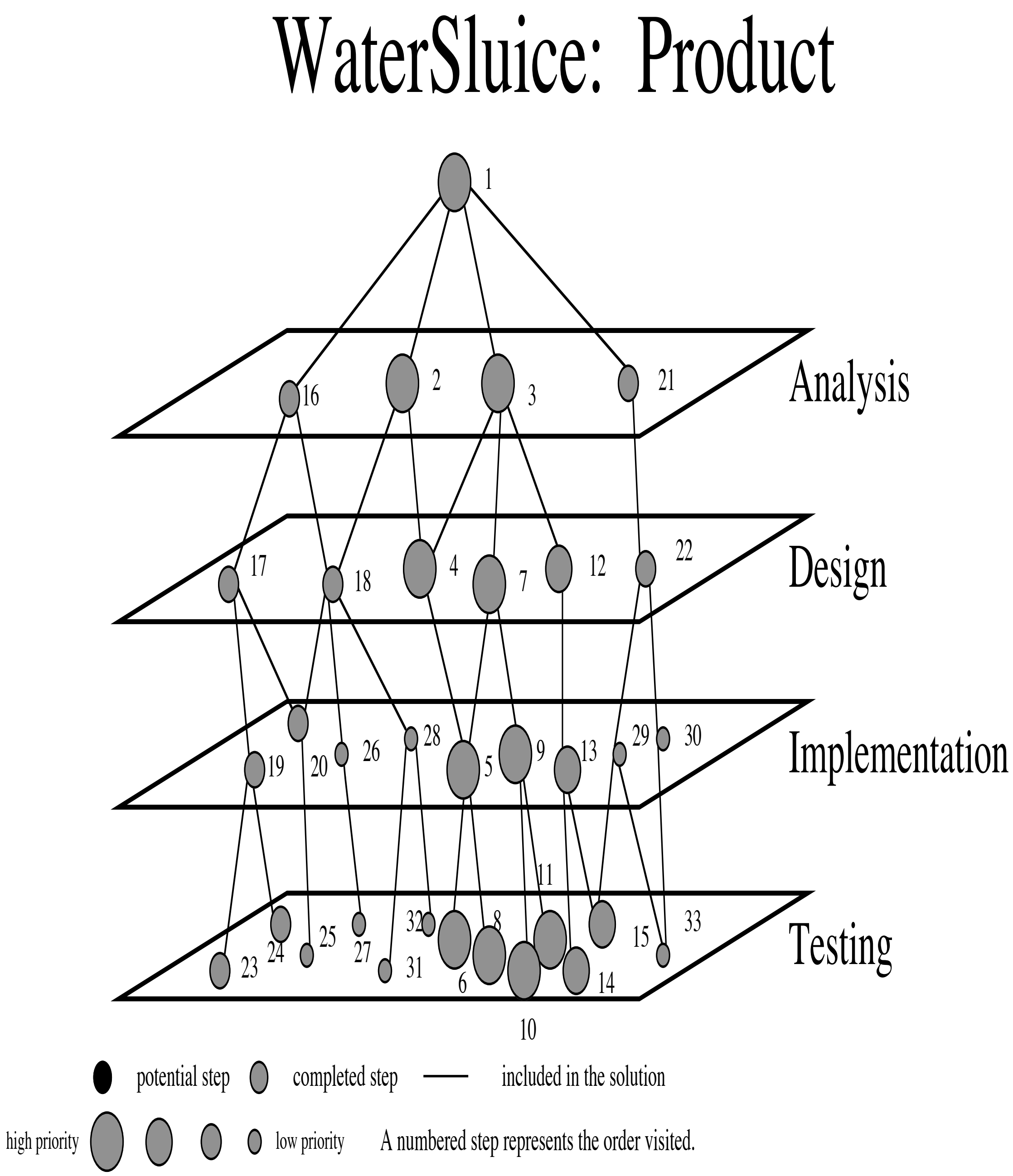


Figure 5.18: WaterSluice: Product

step 11*: The methodology is now at the proof of principle stage.*

$$\begin{aligned} PQ &= a_{12} \\ S^{11} &= a_1, a_2, a_3, d_4, i_5, t_6, d_7, t_8, i_9, t_{10}, t_{11} \end{aligned}$$

$\vdots$

step 15*: The methodology is now at the prototype stage.*

$$\begin{aligned} PQ &= a_{16}, a_{21} \\ S^{15} &= a_1, a_2, a_3, d_4, i_5, t_6, d_7, t_8, \\ &\quad i_9, t_{10}, t_{11}, d_{12}, i_{13}, t_{14} \end{aligned}$$

$\vdots$

step 24*: The methodology is now at the alpha stage.*

$$\begin{aligned} PQ &= i_{26}, i_{28}, i_{29}, i_{30} \\ S^{15} &= a_1, a_2, a_3, d_4, i_5, t_6, d_7, t_8, \\ &\quad i_9, t_{10}, t_{11}, d_{12}, i_{13}, t_{14}, \\ &\quad t_{15}, a_{16}, d_{17}, d_{18}, i_{19}, i_{20}, \\ &\quad a_{21}, d_{22}, t_{23}, t_{24} \end{aligned}$$

$\vdots$

step 32: The methodology is now at the beta stage.

$$
\begin{aligned}
PQ &= t_{33} \\
S^{15} &= a_1, a_2, a_3, d_4, i_5, t_6, d_7, t_8, \\
&\quad i_9, t_{10}, t_{11}, d_{12}, i_{13}, t_{14}, \\
&\quad t_{15}, a_{16}, d_{17}, d_{18}, i_{19}, i_{20}, \\
&\quad a_{21}, d_{22}, t_{23}, t_{24}, \\
&\quad t_{25}, i_{26}, t_{27}, i_{28}, \\
&\quad i_{29}, i_{30}, t_{31}, t_{32}
\end{aligned}
$$

$\vdots$

step $n_a + n_d + n_i + n_t$*: The priority queue contains the last step of the final system acceptance test.*

Thus, the WaterSluice software engineering methodology is static complete.

The WaterSluice software engineering methodology uses priority to guide the process combined with the iterative mechanism found in a cyclical methodology and governed by the steady progression found in a sequential methodology. If the environment is static, then all steps in the environment are known before the WaterSluice software engineering methodology begins. The WaterSluice software engineering methodology first discovers high priority steps from the analysis plane, the design plane, the implementation plane, and the testing plane. As higher priority steps are discovered and visited, they are eventually exhausted. This allows the lower priority steps to be visited. The WaterSluice software engineering methodology continues until the only remaining step to visit is the lowest possible priority step.

Corollary 4.1 (WaterSluice: Dynamic Complete) *The WaterSluice software engineering methodology is dynamic complete.*

Proof 4.1.1 *The WaterSluice software engineering methodology allows for the introduction of new information at every step and the removal of information that is no longer needed. Recall the presence of the priority queue. When steps are placed on the priority queue, the queue is rearranged. Low priority steps migrate to the end of the queue while high priority steps migrate to the beginning of the queue. Regardless of when a high priority item is discovered, the methodology will be able to react. For a dynamic environment, the WaterSluice software engineering methodology will eventually find the solution.*

Corollary 4.2 (WaterSluice: Non-monotonic Complete) *The WaterSluice software engineering methodology is non-monotonic complete.*

Proof 4.2.1 *The WaterSluice software engineering methodology allows for the introduction of new information at every step and the removal of information that is no longer needed. Define a priority function that takes into account non-monotonic steps. Consistent steps are assigned similar high priority. Non-consistent steps are assigned lower priority. When steps are placed on the priority queue, the queue is rearranged. Low priority steps migrate to the end of the queue while high priority steps migrate to the beginning of the queue. Regardless of when a high priority item is discovered, the methodology will be able to react. For a non-monotonic space, the WaterSluice software engineering methodology will eventually find the solution leaving behind the conflicting steps.*

Corollary 4.3 (WaterSluice: Best Case) *The best case performance of the WaterSluice software engineering methodology is O(1).*

Proof 4.3.1 *Let the environment consist of n_a analysis steps, n_d design steps, n_i implementation steps, and n_t testing steps. Let the correct analysis step be the very first step a_1. Let the correct design step be the very first step d_1. Let the correct implementation step be the very first step i_1. Let the system acceptance test be the very first step t_1. To satisfy the problem statement would require one analysis, one design, one implementation, and one testing step. In this environment the WaterSluice software engineering methodology will require only four steps. Thus the performance is O(1).*

Corollary 4.4 (WaterSluice: Worst Case) *The worst case performance of the WaterSluice software engineering methodology is O(N) where N is the size of the environment.*

Proof 4.4.1 *Let the environment consist of n_a analysis steps, n_d design steps, n_i implementation steps, and n_t testing steps. Let the system acceptance test be the lowest priority step t_{n_t}. The number of steps to find a solution is $n_a + n_d + n_i + n_t$ which is O(N).*

Corollary 4.5 (WaterSluice: Average Case) *The average case performance of the WaterSluice software engineering methodology is an order-of-magnitude less than N where N is the size of the environment.*

Proof 4.5.1 *Let d be the depth of the space. In the special case of this space d is 4. Let b be the average fan out of a step in the space if there was no priority function. If the space has N steps then*

$$N = b^0 + b^1 + b^2 + \cdots + b^d.$$

To a reasonable approximation, only the last term is significant and

$$N \approx b^d.$$

Consider the addition of a priority function. The best case priority function would guide the algorithm directly to a solution. In this case, the fan out would be 1 and the total number of steps would be O(1). A worst case priority function would not guide the algorithm. Thus the fan out would be b and the total number of steps would be O(N). In the average case, the priority function would trim the fan out by half. Thus

$$\begin{aligned} N_{average} &\approx \left(\frac{b}{2}\right)^d \\ &\approx \frac{b^d}{2^d} \\ &\approx \frac{N}{2^d} \end{aligned}$$

On average the performance of the algorithm is an order-of-magnitude less than N.

Methodology	Static Complete	Dynamic Complete	Non-monotonic Complete
sequential	yes	no	no
cyclical	yes	yes	no
WaterSluice	yes	yes	yes

Table 5.1: Summary of Completeness

Methodology	Best	Worst	Average
sequential	$O(N)$	$O(N)$	$O(N)$
cyclical	$O(1)$	$O(N)$	$O(N)$
WaterSluice	$O(1)$	$O(N)$	order-of-magnitude less than N

Table 5.2: Summary of Performance

5.4 Summary Results from the Main Theorem

Theorem 1 *Different software engineering methodologies have significant performance variations depending on the given environment. A software engineering methodology that is goal focused, manages conflicts, and differentiates between different priorities is best suited for dynamic non-monotonic environments.*

The proceeding theorems and corollaries generate several key results:

- All three categories of software engineering methodologies are complete for static environments. See Theorems 2, 3, and 4.
- Only cyclical and WaterSluice are complete for dynamic environments. See Corollaries 2.4, 3.4, and 4.1.
- Only WaterSluice is complete for non-monotonic environments. See Corollaries 2.5, 3.5, and 4.2.
- The best case performance of sequential software engineering methodology is O(N). See Corollaries 2.1.

- The best case performance of cyclical and WaterSluice software engineering methodologies is O(1). See Corollaries 3.1, and 4.3.

- The worst case performance of all three categories of methodologies are the same. See Corollaries 2.2, 3.2, and 4.4.

- On average, the sequential methodology will find a solution in O(N). See Corollary 2.3. On average, the cyclical methodology will find a solution in O(N). [5] See Corollary 3.3. On average, the WaterSluice will find a solution an order-of-magnitude less than N. See Corollary 4.5.

The observations are summarized in Table 5.1 on page 114 and Table 5.2 on page 114.

[5] A more accurate average performance measurement for a cyclical software engineering methodology is N/2.

Chapter 6

An Analogy with Search

The formal foundation chapter presented results on methodologies and their performance in various environments. These results are analogous to similar results from search theory. See [82], [84], [83], and [98].

6.1 Search Background

There are three well-known search algorithms: **breadth-first**, **depth-first**, and **best-first**. The basic background of these three algorithms are presented along with their accompanying search space.

In a **breadth-first** algorithm, the search is concentrated at the high level and not until a solution is found at this level does the algorithm go deeper into the lower levels. This algorithm is queue-based, and almost the entire search space needs to be searched before an answer is found. On the average, this algorithm is O(N) in complexity where N is the number of nodes in the tree representing the search space. Best case performance for a breadth-first algorithm is and worst case performance are O(N).

In a **depth-first** algorithm, the search is concentrated at the lower levels. This algorithm is stack-based, and a potential solution may be found early in the search. The worst case performance is no better than a breadth-first algorithm, but on average a depth-first algorithm will find a feasible solution quicker. On the average, this algorithm is O(N) in complexity where N is the number of nodes in the tree representing the search space. Best case performance for a depth-first algorithm is O(1), while worst case performance is O(N).

The depth-first algorithm has a problem around sections of the tree that represent near solutions. The algorithm will get stuck on a local optimum and not find the best solution until much later in the search. This problem is called hill climbing.

In a **best-first** algorithm, the search is concentrated on the next best move. All next moves are prioritized by looking one move ahead and only the next best move is taken. After each move, additional moves may be possible and are added to the list of candidates. The process continues until an optimum solution is found. The search space is searched in a jumping fashion as the algorithm hops between different areas of higher interest. This algorithm is based on a priority queue that is usually based on a partial order tree.

The best known of the best-first algorithms is called A*. The priority functions are split into two components. One represents the known cost to get to a node in the search space while the other represents the estimated cost of continuing towards the goal. The estimated cost must be positive and must be an underestimation of the actual cost. It can be shown that A* is the best of all best-first search algorithms. The A* algorithm is more complex because it requires the definition of the priority function. On the average, this algorithm is an order-of-magnitude less than N in complexity where N is the number of nodes in the tree. Best case performance for a best-first algorithm is O(1), while worst case performance is O(N).

There is an interesting tradeoff between the cost of visiting a node in the search space and the cost of calculating the priority function. If the search space is small, inexpensive to traverse, and the cost of calculating the priority function is expensive, then the depth-first and breadth-first algorithms may have better total performance over the best-first algorithm. The cost of calculating the priority function can be controlled by varying the quality of the answers returned by the priority function. If the search space is complex and large, then the cost of calculating a precise priority function is negligible. On the other hand, some situations call for a cheap priority function. In the limiting case, the priority function could be simply that all next steps have the same priority and the algorithm becomes a breadth-first algorithm. Alternatively, the priority function could reflect the depth of the search space and the best-first algorithm would behave like a depth-first algorithm.

These search algorithms use a search space. A search space consists of a collection of nodes or states. There are two special states called initial and goal. There is a function that walks the search space using the primitive *next step*. Optionally, the states may be labeled for later reference. The path from initial state to the goal state is called the solution. Between a state and its reachable next states are associated costs. Only the best-first algorithm uses this cost information for other than summation or report generation reasons. In general, the algorithms produce a directed acyclic graph as a result of the search.

6.2 Analogy: Search and Methodologies

These search algorithms and their accompanying search space can be extended to apply to software engineering methodologies. By analogy, a sequential methodology can be compared to a breadth-first search algorithm, a cyclical methodology to a depth-first search algorithm, and the WaterSluice methodology to a best-first search algorithm.

A solution in search space is the path from the initial node to the goal node. Many nodes visited may not be included in the solution path. On the other hand, a solution in software engineering methodology space is the entire DAG necessary to go from the initial problem statement to the final acceptance test.

Since the space is much larger than the solution path it is imperative to prune the search as much as possible. Pruning is affected by dynamic and non-monotonic considerations when the entire search space cannot be pre-composed.

6.3 Conclusion

The results in this thesis on methodology performance are analogous to the associated results in search theory. All three search algorithms are complete for a static search space. The differences appear when the search space is dynamic. A breadth-first search algorithm may miss a solution. Both a depth-first search algorithm and a best-first search algorithm will find a solution in a dynamic search space. In some cases, the best-first search algorithm will find a solution in less than or equal time to the other two methodologies. The worst case performance of all three search algorithms are the same. On average, the breath-first search algorithm will find a solution in O(N). On average, the depth-first search algorithm will find a solution in O(N). On average, the best-first search algorithm will find a solution is an order-of-magnitude less than N.

Chapter 7

Project Surveys

Since a realistic quantifiable experiment on software engineering methodologies cannot be carried out, this chapter presents projects from the author's and colleagues experiences. These experiences help formulate and substantiate the formal work of this thesis.

7.1 Introduction

In this chapter, a survey form is presented. This survey form will help guide the classification of various software engineering methodologies and their usage in a variety of projects. The survey form is then completed for the three main categories of sequential, cyclical, and WaterSluice software engineering methodologies. Survey forms for the Boehm-waterfall and the Boehm-spiral are also completed. Several software engineering projects are then presented along with their software engineering methodology and completed survey form. Because there is a very limited number of publications of real-life software engineering methodology usage, most of the example projects are from the author's experiences. Several projects are from well known systems such as Ada, UNIX, and X.

This survey is intended as an aid in understanding software engineering methodologies. This survey is not presented as the ultimate, definitive, scientific classification schema for software engineering methodologies.

7.2 The Survey

The survey is a series of questions with a choice of alternative answers. The survey is also presented in a summary tabular form. An accompanying information page helps to clarify some of the questions and response options.

7.2.1 Software Engineering Methodology Phases

The first section of the survey deals with methodology phases and their usage. There are four main phases of analysis, design, implementation, and testing. The analysis phase establishes the requirements. The design phase establishes the architecture. The system is built in the implementation phase while quality is assured in the testing phase. See the sections on analysis, design, implementation, and testing in table 7.1 on page 124.

7.2.2 Software Engineering Methodology Composition

The next section of the survey deals with the composition of the four main phases. ADIT is an acronym for analysis, design, implementation, and testing composed in that order. There may be many cycles of ADIT.

In each phase, the alternatives may be prioritized. Change order control is a process used to resolve conflicts. These conflicts may be non-monotonic in nature, where taking one action negates actions already accomplished. The ADIT cycles may be used to create versions of the system where several baselines of the system are established with potentially each baseline having more functionality. A baseline of the system may be released in the sequence of internal prototype, external prototype, alpha, and then beta releases. See the sections on cycles, priority, versions, and change control in table 7.1 on page 124.

7.2.3 System Size Estimates

The next section of the survey deals with system size estimates in total number of person years to build the system and in the calendar time. The person years may include all work in analysis, design, implementation, and testing but excludes customer usage. See the sections on duration and effort in table 7.1 on page 124.

7.2.4 Non-monotonic Characteristics

The last section of the survey deals with the non-monotonic characteristics of the software engineering methodology. See table 7.2 on page 124.

7.2.5 The Tabular Form

The tables 7.1 and 7.2 starting on page 124 represent the questions in tabular form.

Question	Response
Analysis?	none..poor..fair..good..great
Design?	none..poor..fair..good..great
Implementation?	none..poor..fair..good..great
Testing?	none..poor..fair..good..great
Cycles of ADIT?	one..few..several..frequent
Priority?	no..yes
Versions?	no..yes
Change Order Control?	no..yes
Internal Prototype?	no..yes
External Prototype?	no..yes
Alpha Release?	no..yes
Beta Release?	no..yes
Duration?	number..
Effort?	number..

Table 7.1: Survey Part 1: Basic Properties

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..seldom..often..frequent
Did an architectural change ever negatively affect accomplished work?	never..seldom..often..frequent
Are new features introduced up to product release?	never..seldom..often..frequent
Is there a dedicated period of quality assurance before the product is released?	no..yes

Table 7.2: Survey Part 2: Change Control

7.3 Projects

7.3.1 TDS Health Care System

The TDS [92] health care system supported many health care applications including order entry, result reporting, medication tracking and scheduling, vital signs, flow sheets, active patient care, nursing support systems, day-to-day management of a patient while in the hospital, short term medical information, laboratory orders and result reporting, pharmacy, and day-to-day charting.

This system was built by Lockheed in the mid 1960s as a spin off of the space program. Congress wanted to show that the same technology advancement that landed a person on the moon could also have down to earth applications. The first customer was a medium sized local hospital: El Camino Hospital.

The original Lockheed team had several hundred members and it took several years to build the base system. It took several more years of hospital testing to make the system useful and to establish high quality.

The architecture was tool based. In this tool based architecture, the system was built around several authoring tools. One tool was used to author formularies and list. Another tool was used to author application logic and presentation, while another tool was used to author database records. A general purpose database and user interface engine was provided. The content was authored by the tools and then interpreted by the database. These and the user interface engine defined the application.

The applications were split into two halves. One half of the application defined the database while the other half of the application defined the application logic and the user interface. In fact, it would be more accurate to say that the user interface defined the application logic. This was a client/server architecture with fat clients. Note that the GUI was displayed on dumb monitors and the client ran on the mainframe.

The database, called M1, was influenced by MUMPS. The underlying structure of the database resembled artificial intelligence frames. Every record could have a variable number of fields with each field having a facet. A facet defines how the field of the record is to be obtained. A value facet would contain the data. A function facet would contain an algorithm to calculate the data. Other facets included domain checks, pre-conditions, post-condition, format information, and triggers. The trigger facet was used to notify other associated records that a change in this data record has occurred. Records need not have the same scheme nor be complete. A tool was provided to manage the database. Features included the ability to view, update, delete, and create the data.

The user interface, called M2, was also frame-based. One frame equals one screen. The fields on the screen were controlled by the facets in the frames. These facets governed display information, application logic, entry format constraints, default values, and screen navigation. A customer would click through screens and complete a database frame. Once the frame was completed, the customer would save the frame. The act of saving the frame may trigger other actions. A tool was provided to build and manage frames.

The system had several technology innovations for the mid 1960s. These included customixed light-pen, light-sensitive monitors, and networking cards. The system was hand-crafted in assembly code.

Included with the system was an example hospital. This included several thousand screens which represented several dozen health care applications. The system took about a thousand person years to build and stabilize for the first hospital. Several more thousands of person years were invested to clone the system into about 100 other hospitals. Since this system was very expensive, millions of dollars per hospital per year, only the largest hospital could afford such an investment. Each customer required extensive training of several person months.

Once the system was built, many other hospitals could be cloned from the original. Hospitals, and health-care in general, are about the same. They differ on details of content but not so much on difference of functions. For example, every hospital has a pharmacy where prescriptions are filled but the exact formulary differs greatly.

El Camino Hospital played a critical role as the very first customer. El Camino Hospital provided high quality domain knowledge and acted as testers. By having the system active in a working hospital gave credibility to the system and raised the level of trust.

This system was stable for thirty years. The tool-based architecture allowed for new content, new screens, and new application logic. The frame-based system allowed for flexibility of information. After initial development, the system was moved from a NASA project to a company. Individuals were given the option to follow the project or to stay in the company.

The software engineering methodology was cyclical. Small changes were introduced, tested in the example hospital, released to the customer, errors were incrementally fixed, and new applications were incrementally developed.

The original team did not generate a requirement document nor an architecture document. The original key individuals did not follow the system into productization. The remaining team focused on content while the underlying tools and infrastructure remained stagnate. Technology changes soon made the underlying infrastructure obsolete. Attempts to change the infrastructure had the affect of converting the code from a stable base to a fragile tangle of spaghetti code. The product disappeared in the late 1990s.

The tool-based architecture along with the underlying frame-based data model allowed the product to reflect changes in content. The custom hardware, lack of requirement and architectural knowledge, hand-crafted assembly code, and loss of key individuals contributed to the product obsolescence.

The cyclical software engineering methodology generated a well focused, hospital-base, short-term clinical information, nursing system but missed the bigger pictures of including doctors, administration, and the electronic patient chart.

See tables 7.3 and 7.4 starting on page 128.

Question	Response
Analysis?	none..**poor**..fair..good..great
Design?	none..poor..fair..good..**great**
Implementation?	none..poor..**fair**..good..great
Testing?	none..poor..**fair**..good..great
Cycles of ADIT?	one..few..several..**frequent**
Priority?	**no**..yes
Versions?	no..**yes**
Change Order Control?	**no**..yes
Internal Prototype?	**no**..yes
External Prototype?	**no**..yes
Alpha Release?	**no**..yes
Beta Release?	**no**..yes
Duration?	**35 years**
Effort?	**greater than 5,000 person years**

Table 7.3: Survey Part 1: Basic Properties TDS

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..seldom..often..**frequent**
Did an architectural change ever negatively affect accomplished work?	never..seldom..often..**frequent**
Are new features introduced up to product release?	never..seldom..often..**frequent**
Is there a dedicated period of quality assurance before the product is released?	**no**..yes

Table 7.4: Survey Part 2: Change Control TDS

7.3.2 Digital's Virtual Memory System (VMS)

VMS ([47], [80], [46]) is an operating system build for the (Virtual Address eXtension) VAX instruction set hardware in the late 1970s. The operating system had a strong following in the 1980s with a decline of usage in the 1990s. The original team was very small, only a dozen or so very experienced individuals. The team quickly grew to several hundred, and remained at that level for over a decade. For performance reasons, the product was built in highly-structured assembly code. VMS was materialized by tens of millions of lines of code and took days to compile. VMS was built to be the leading-edge technology operating system targeted at leading-edge technology customers. People would say, "Digital was an engineering company, run by engineers, for engineers." One only had to make a technology argument to add a new feature to VMS. The Digital engineers built the system for themselves and then shared the system, for a fee, with others.

VMS was one of the first major operating systems to support virtual memory, 32 bit address space, 128 bit floating point number precision, and a complex instruction set.

The software engineering methodology was best-first similar to the WaterSluice. The requirement committee would prioritize requirements and establish a list of requirements to be placed in the next version. For the most part, this list would be placed under change control. The architecture group, the small core of original developers, would establish the design and high level development plans.

The engineering teams would implement the architecture in a two week cycle. One week, called the red week, new development would be added to the code base. On Friday, the code would be frozen and over the weekend the regression suites would test the new features and make sure that already established features would not break. The next week, the blue week, the engineering teams concentrated on bug fixes. This short cycle consisting of build-a-week, followed by test-a-week, would quickly converge to the next version of the operating system.

A large regression test suite was maintained. Not only testing for operating system features, but the regression testing of all applications, numbering several hundred, were also included. One of the best indications of a stable operating system are stable applications on top of the operating system.

The engineering team would always be using their current build. Several months before customer alpha release, the corporation would be placed on the new release. An alpha customer would get a product that has been deployed for several months to thousands of machines. The beta release would include bug fixes to the alpha release. By the time the new version was released it was already relatively high in quality.

If you include development and internal testing, both regressing testing, and internal operational testing, each year thousands of person years were involved in VMS releases and development.

The version release of the product reflected the weekly two phase cycle. Versions ending in even numbers, 1.0, 1.2, 1.4, etc., had new features while versions ending in odd numbers, 1.1, 1.3, 1.5, etc., had bug fixes. The odd versions actually included the new code for the next even version, but this code was disabled or running in shadow mode only. Just the presence of new code, even disabled, would introduced bugs associated with memory management, locking, and race conditions.

There were several decisions which eventually lead to the decrease in popularity of VMS. First, the VAX instruction set was a member of the complex instruction set families. In the late 1970s, instruction sets were getting more and more functionality. This made the compilers easy to write. A compiler was not much more then a pattern matcher with rewrite rules. As compilers got smarter, this allowed for instruction sets to become simpler, and a new family, the RISC, of simple instruction sets was introduced. This was welcome news to the hardware makers because it let the hardware developers concentrate on speed and performance and let the compiler handle complexity of translating algorithms and data structures into sequences of simple instructions.

Second, VMS was written in VAX assembly language. This made translating the millions of line of code a daunting task.

Third, architecture design of VMS had a major flaw. An operating system needs to lock critical sections. In fact, the correctness of just about every line of code is highly influenced by correct locking. VMS used a highly non-standard feature of interrupt priority levels to get the affect of locking. There were only 32 such levels, leading to only 32 major locks. The physical lock table was only 32 by 32 while the effective lock table was more like thousands by thousands. Every lock was highly overloaded in meaning. This did not scale and made VMS more complex. After a decade of development, VMS was moved to the alpha chip set, but only after the alpha chip set was modified to support interrupt priority levels. Had the original team separated out the locking mechanism to a more general architecture, this dependency could have been avoided.

Fourth, only a few people understood the total picture of the locking scheme. The effective lock table for VMS was in their heads. This knowledge was never really captured in writing and really reflected years of experience of working with the architecture of VMS.

See tables 7.5 and 7.6 starting on page 132.

Question	Response
Analysis?	none..poor..fair..**good**..great
Design?	none..poor..fair..**good**..great
Implementation?	none..poor..fair..**good**..great
Testing?	none..poor..fair..good..**great**
Cycles of ADIT?	one..few..several..**frequent**
Priority?	no..**yes**
Versions?	no..**yes**
Change Order Control?	no..**yes**
Internal Prototype?	no..**yes**
External Prototype?	no..**yes**
Alpha Release?	no..**yes**
Beta Release?	no..**yes**
Duration?	**20 years**
Effort?	**greater than 10,000 person years**

Table 7.5: Survey Part 1: Basic Properties VMS

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..**seldom**..often..frequent
Did an architectural change ever negatively affect accomplished work?	never..**seldom**..often..frequent
Are new features introduced up to product release?	never..**seldom**..often..frequent
Is there a dedicated period of quality assurance before the product is released?	no..**yes**

Table 7.6: Survey Part 2: Change Control VMS

7.3.3 Stanford University Infrastructure

The business functions of Stanford University are currently managed by a relatively large group of developers and maintainers. The group numbers about 200. In the early 1970s, this group began building an environment based on SPIRES ([94], [93], [104]) a hierarchical text database system originally used to store scientific (high energy physics) documents. On top of SPIRES was built an infrastructure to support email, event messaging, work flow, forms routing, and digital signature. Using this infrastructure, the two major applications, one centered around accounting and the other centered around students information, were written.

The development of this system took about two and a half decades. This is an example of a thousands of person year project.

The software engineering methodology followed was cyclical in nature. The builder's of the system and the customer's of the system were almost one in the same. Small incremental changes were made to meet small requirement changes. Testing was done by the staff at the help desk. If it worked on the examples in the help manual, it was declared to be ready for deployment. Deployment was easy because it was only on one machine. Customers would connect through terminals. Everything was custom built to handle small-detailed changes.

There is no one requirement document. There is no one architecture document. There is no one person who understands the complete system. This is a recipe for pending disaster. Individual members in the development team knew only local information.

The system served the university well until the wake-up call in the mid 1990s. The federal government made charges against the university of major errors in billing. To show otherwise would require a special query across the general ledger. This class of queries was never before attempted and had to be written, in assembly code, from scratch. It took over a year and about 10 person years worth of work to finish this query that showed the university to be in compliance.

See tables 7.7 and 7.8 starting on page 135.

Question	Response
Analysis?	none..**poor**..fair..good..great
Design?	none..poor..fair..**good**..great
Implementation?	none..poor..fair..**good**..great
Testing?	none..poor..**fair**..good..great
Cycles of ADIT?	one..few..several..**frequent**
Priority?	**no**..yes
Versions?	no..**yes**
Change Order Control?	**no**..yes
Internal Prototype?	**no**..yes
External Prototype?	**no**..yes
Alpha Release?	**no**..yes
Beta Release?	**no**..yes
Duration?	**25 years**
Effort?	**greater than 1,000 person years**

Table 7.7: Survey Part 1: Basic Properties Stanford

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..seldom..**often**..frequent
Did an architectural change ever negatively affect accomplished work?	never..seldom..**often**..frequent
Are new features introduced up to product release?	never..seldom..often..**frequent**
Is there a dedicated period of quality assurance before the product is released?	**no**..yes

Table 7.8: Survey Part 2: Change Control Stanford

7.3.4 Independent Technology Inc. (ITI)

ITI built client-server applications using key components of networking, transaction processing, graphical user interfaces, relational databases, UNIX, and object-based programming. Their architecture paradigm was to build a common interface to the key components. This common interface could then be layered on a particular vendor's solution. In this way, an application built on this common interface could easily be migrated from similar but disparate vendor provided solutions. Hence the name of the company: independence.

The biggest product built was a health insurance claims processing system. Internal claims processing clients would process claims. The claims would be routed through the system using a work flow graph. A claim would be represented as relational data and images of the original form. The system was deployed to about a thousand clients talking to a multi-processor server. This system took 50 person years to build the tool kit and 15 person years to build the application.

The software engineering methodology used was cyclical. Small incremental changes to the application were done, shown to the customer, and iterated upon. There was no requirement document except for a very general one-page statement. The architecture was well defined by the tools.

What was missing was a total lack of quality assurance. Less than 0.5 person years was spent on testing before the product went live at the customer's site. Needless to say, the system was not reliable and not well accepted. Now the company had to concentrate on doing nothing but bug fixes. With no formal regression testing suite, each bug fixed would most likely create, or uncover, another bug. After several months, the system was rejected by the customer and the company soon went out of business.

The individual responsible for quality assurance would generate reports with the same conclusion: performed as expected. To a casual reader this meant high quality with no real problems. However, the quality assurance person meant something entirely different. He expected low quality and found low quality. Hence the report: performed as expected.

The tool kit was sold. Having gone through the first application, the tool kit was some what real. The tool kit was also later abandoned. The architectural goal of independence was not accomplished. The applications were independent of vendor specific systems, but now the application was dependent on the tool kit. In fact, the tool kit was just another vendor. A better approach to independence would have been to use accepted standards as the foundation of the tool kit.

See tables 7.9 and 7.10 starting on page 137.

Question	Response
Analysis?	none..**poor**..fair..good..great
Design?	none..poor..**fair**..good..great
Implementation?	none..**poor**..fair..good..great
Testing?	**none**..poor..fair..good..great
Cycles of ADIT?	one..few..several..**frequent**
Priority?	**no**..yes
Versions?	**no**..yes
Change Order Control?	**no**..yes
Internal Prototype?	**no**..yes
External Prototype?	no..**yes**
Alpha Release?	**no**..yes
Beta Release?	**no**..yes
Duration?	**three years**
Effort?	**about 50 person years**

Table 7.9: Survey Part 1: Basic Properties ITI

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..seldom..often..**frequent**
Did an architectural change ever negatively affect accomplished work?	never..seldom..often..**frequent**
Are new features introduced up to product release?	never..seldom..often..**frequent**
Is there a dedicated period of quality assurance before the product is released?	**no**..yes

Table 7.10: Survey Part 2: Change Control ITI

7.3.5 Oceania

Oceania is a health care company founded by physicians. The physicians worked in the emergency trauma center on weekends pulling a 48 hour shift. The income derived from this endeavor was more than enough to support themselves and to fund Oceania.

Oceania hired about a dozen college experienced but non-graduate junior-implementors. This team, over a five year period, generated an impressive demo of a physician-centric health care system based on the NeXT machine. The demo was impressive enough to obtain venture capital investment and three signed customers. An additional year passed with no shipped product; an experienced software engineer was needed.

There was no requirement document, no architecture document, and no testing paradigm. In fact, the system only contained the GUI. There was no database and no application logic. No wonder there was no product.

This situation is actually understandable. For the most part, customers are very

GUI centric. What the customer sees on the screen, is the customer's total view of the rest of the application. The underlying infrastructure is assumed to exists if the GUI is present. A junior engineering team, lead by GUI-centric physicians, would build the best GUI in the world and not even realize the massive amount of missing code.

The software engineering process in place was cyclical in nature, only iterating on the GUI. This generated a wonderful GUI that could not be supported.

A more formal software engineering process was put in place. A requirement document and architecture were established for the GUI, the application logic, and the database components. A testing plan was put in place. The scope of the application was grossly reduced. The signed customers were replaced with a hospice. In this way, any error in the system would not have any affect on the outcome of any patient, since all patients in the hospice are terminally ill. The product was shipped and went live and was active for over a year without any customer discovered bugs.

The engineering process was turned back into the control of the physicians. Unfortunately, the physicians quickly reverted to their established previous behavior and the company had difficulties in delivering another successful deployed product.

There is a fundamental difference between the training of a physician and the training of an engineer. In the practice of health care, a physician must always base his decisions on the most current known information and protocols. As new information is discovered about the patient and as new procedures are established, the corrective actions that a physician takes are different than the corrective actions based on the outdated information. A physician acts in a non-monotonic fashion. Current information is much more important than past historical information.

An engineer, on the other hand, acts in a monotonic fashion. Actions are based on facts. Work accomplished so far should not be thrown out without significant reasons. A physician wants to base all actions on the most current information. Engineers need to base actions on all information, both current and past.

The conflict between the two paradigms was the root cause of major problems for Oceania. In particular, when one relational database vendor came out with a new product, the physicians wanted to immediately change over and throw out the work done on the other relational databases vendor product. The engineers agreed that the new release was better than the old release, but why throw out all of that work just for a little gain.

An important missing process concept in software engineering methodology was apparent. Non-monotonic changes to a system needs to be carefully managed. Some of the changes are necessary and should be allowed. Many of the changes are not important enough to delay the shipping of the product. Even if the product does not have the best current answer to all problems, a good answer is usually good enough. See tables 7.11 and 7.12 starting on page 141.

Question	Response
Analysis?	none..**poor**..fair..good..great
Design?	none..**poor**..fair..good..great
Implementation?	none..**poor**..fair..good..great
Testing?	none..**poor**..fair..good..great
Cycles of ADIT?	one..few..several..**frequent**
Priority?	**no**..yes
Versions?	**no**..yes
Change Order Control?	**no**..yes
Internal Prototype?	no..**yes**
External Prototype?	**no**..yes
Alpha Release?	**no**..yes
Beta Release?	**no**..yes
Duration?	**over 10 years**
Effort?	**100 person years**

Table 7.11: Survey Part 1: Basic Properties Oceania

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..seldom..often..**frequent**
Did an architectural change ever negatively affect accomplished work?	never..seldom..often..**frequent**
Are new features introduced up to product release?	never..seldom..often..**frequent**
Is there a dedicated period of quality assurance before the product is released?	**no**..yes

Table 7.12: Survey Part 2: Change Control Oceania

7.3.6 CONMOD

CONMOD stands for CONflict MODeling and was a government program to simulate the battlefield. One battle could be simulated with different weapons, strategies, soldiers, weather, and battle fields without the needless destruction of resources and at a fraction of the cost.

The project was a follow on project to JANUS. JANUS had been around for several decades and resisted many efforts to modernize.

CONMOD would be based on the technology of objects, Ada, relational databases, large monitors with color GUIs, Digital's VAX computers running VMS, and expert systems.

There was no real software engineering methodology followed. There was no requirement or architecture document. There was no testing paradigm. The military would place about a dozen high level professional officers on a 12 month rotation. The civilian programmers would get their daily work assignments from the officers.

A significant amount of time was spent on the random number generator. Since this was a discreet simulation where every action would create an event with a probabilistic outcome, the random number generator was considered high priority. Every actor on the battle field, be it personnel or munitions, would have an event queue. As time progressed, actions would trigger events. Everything was to be modeled except for the command and control. Command and control would be provided by military officers guiding the simulation.

CONMOD was to be a non-classified project using only military information gathered from public sources. The military officers would have a private session on what information to share with the civilians.

The project never accomplished more than a very simple simulation. There was a large gap on communication between the civilian programmers who wanted to talk algorithms and data structures and the military officers who wanted to talk about military campaigns. The project lasted five years with about 100 person years of effort invested. See tables 7.13 and 7.14 starting on page 143.

Question	Response
Analysis?	none..**poor**..fair..good..great
Design?	none..**poor**..fair..good..great
Implementation?	none..**poor**..fair..good..great
Testing?	none..**poor**..fair..good..great
Cycles of ADIT?	one..**few**..several..frequent
Priority?	**no**..yes
Versions?	**no**..yes
Change Order Control?	**no**..yes
Internal Prototype?	no..**yes**
External Prototype?	**no**..yes
Alpha Release?	**no**..yes
Beta Release?	**no**..yes
Duration?	**5 years**
Effort?	**100 person years**

Table 7.13: Survey Part 1: Basic Properties CONMOD

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..seldom..often..**frequent**
Did an architectural change ever negatively affect accomplished work?	never..seldom..often..**frequent**
Are new features introduced up to product release?	never..seldom..often..**frequent**
Is there a dedicated period of quality assurance before the product is released?	**no**..yes

Table 7.14: Survey Part 2: Change Control CONMOD

7.3.7 UNIX

UNIX ([124], [123]) is an operating system originating at AT&T Bell Labs in the 1970s. UNIX was one of the first operating systems written in the high level language, C, and intended to be machine hardware independent. Many versions of UNIX exist and the influence of UNIX on other operating systems is dramatic.

Versions of UNIX included virtual memory, multi-processing, symmetric multiprocessor, a file system, networking, the X windowing systems, and a script-based user interface shells. Many versions of UNIX are free or near free and the source code is readily available.

UNIX is very popular in university and research environments because of the low cost, advanced features, and readily available source code. A large customer community has contributed massive number of applications, free for the asking. The UNIX distribution includes thousands of user applications.

The key architectural feature that has allowed UNIX to last for such a long time is the communication subsystem design. Everything in the communication subsystem has an index entry called the inode. Given an inode entry, an application can read and write bytes of data. The inode entry may be associated with a network, file system, a process, or a keyboard. Inodes give UNIX applications hardware I/O device independence and allows for dynamic redirection of I/O. A new I/O device is easy to install. Just create an inode, write a device driver, and almost like magic, a new I/O device is now on the system.

See tables 7.15 and 7.16 starting on page 145.

Question	Response
Analysis?	none..poor..fair..**good**..great
Design?	none..poor..fair..good..**great**
Implementation?	none..poor..fair..good..**great**
Testing?	none..poor..fair..**good**..great
Cycles of ADIT?	one..few..several..**frequent**
Priority?	**no**..yes
Versions?	no..**yes**
Change Order Control?	**no**..yes
Internal Prototype?	no..**yes**
External Prototype?	no..**yes**
Alpha Release?	no..**yes**
Beta Release?	no..**yes**
Duration?	**30 years**
Effort?	**greater than 10,000 person years**

Table 7.15: Survey Part 1: Basic Properties UNIX

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..**seldom**..often..frequent
Did an architectural change ever negatively affect accomplished work?	never..**seldom**..often..frequent
Are new features introduced up to product release?	never..**seldom**..often..frequent
Is there a dedicated period of quality assurance before the product is released?	no..**yes**

Table 7.16: Survey Part 2: Change Control UNIX

7.3.8 X

X ([109], [132], [118]) is a windowing system built for UNIX but is intended to be machine and operating system independent. X is built on the client/server model. The server side of X resides on the desktop computer and controls the monitor, keyboard, and mouse. The client side of X may reside anywhere on the network. These X clients are called X-applications. A protocol, the X protocol, is used to communicate between X client and X servers.

X is a windowing system and not a user interface paradigm. Motif is the most common accepted user interface paradigm specifying sliders, buttons, basic design of windows, and other widgets.

X was created in the 1980s at MIT. The team was very small lead by Jim Gettys. X is well documented and the source code is free and readily available.

There were two decisions which hampered the wide acceptance of X. The first decision was not to dictate a common look-and-feel. This lead to many different windowing paradigms. Many times competition leads to better answers, but in this case, competition lead to conflicts and interoperability between systems. Eventually, the standards committee for X picked Motif.

The second decision was to break the inode paradigm of UNIX. This meant that X could not be scripted. One application could no longer run another application in a piped manor. Attempts to fix this problem are underway in the TCL scripting language.

See tables 7.17 and 7.18 starting on page 147.

Question	Response
Analysis?	none..poor..fair..**good**..great
Design?	none..poor..fair..**good**..great
Implementation?	none..poor..fair..**good**..great
Testing?	none..poor..fair..**good**..great
Cycles of ADIT?	one..few..**several**..frequent
Priority?	no..**yes**
Versions?	no..**yes**
Change Order Control?	**no**..yes
Internal Prototype?	no..**yes**
External Prototype?	no..**yes**
Alpha Release?	no..**yes**
Beta Release?	no..**yes**
Duration?	**15 years**
Effort?	**100 person years**

Table 7.17: Survey Part 1: Basic Properties X

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..**seldom**..often..frequent
Did an architectural change ever negatively affect accomplished work?	never..**seldom**..often..frequent
Are new features introduced up to product release?	never..**seldom**..often..frequent
Is there a dedicated period of quality assurance before the product is released?	no..**yes**

Table 7.18: Survey Part 2: Change Control X

7.3.9 Ada

Ada ([71], [5]) was created in the late 1970s to solve the problem of the software programming bottle neck. Millions of lines of code needed to be written to support government and commercial needs. Hardware has gotten faster and cheaper each year, but software programming remained labor intensive and expensive. Furthermore, many systems built in the 1950s through the 1970s were nearing their life cycle end and had to be replaced. Because computer science is a relatively new field, solutions to total life cycle management were not abundant. To make maters worse, the computer science environment consists of a huge collection of heterogeneous hardware, software, programming languages, operating system environments, file systems, and database systems.

The hope of Ada was to create one independent environment, written in one language, to support all applications.

Ada is a strongly typed language supporting object-oriented programming, information hiding, modularization, concurrent programming, generalization through generic classes, and a unified error and exception handling system. The hope was that Ada was powerful enough to express a vast number of algorithms and data structures as well as process control and parallelism. In 1983, Ada became a standard.

The development of Ada went though an extensive period of requirement gathering, prototyping, and review. This was followed by huge amounts of funding for compilers and supporting environments. All government contracts were to use Ada as the language of choice.

Unfortunately, Ada did not eclipse the world. The developers of Ada created a system that was difficult to master, hard to program, and un-forgiving to change. Creating a language does not solve the bigger problems of software engineering methodologies where analysis, requirements, design, and architectures are defined. The strongly-typed pointer concept in Ada made many categories of programming difficult, including algorithms in the fields of AI, databases, operating systems, and file systems.

See tables 7.19 and 7.20 starting on page 150.

Question	Response
Analysis?	none..poor..fair..**good**..great
Design?	none..poor..fair..**good**..great
Implementation?	none..poor..fair..**good**..great
Testing?	none..poor..fair..good..**great**
Cycles of ADIT?	one..few..several..**frequent**
Priority?	no..**yes**
Versions?	no..**yes**
Change Order Control?	no..**yes**
Internal Prototype?	no..**yes**
External Prototype?	no..**yes**
Alpha Release?	no..**yes**
Beta Release?	no..**yes**
Duration?	**20 years**
Effort?	**greater than 10,000**

Table 7.19: Survey Part 1: Basic Properties Ada

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..**seldom**..often..frequent
Did an architectural change ever negatively affect accomplished work?	never..**seldom**..often..frequent
Are new features introduced up to product release?	**never**..seldom..often..frequent
Is there a dedicated period of quality assurance before the product is released?	no..**yes**

Table 7.20: Survey Part 2: Change Control Ada

7.4 Software Engineering Methodologies

7.4.1 A Sequential Software Engineering Methodology

See tables 7.21 and 7.22 starting on page 151.

Question	Response
Analysis?	none..poor..fair..good..**great**
Design?	none..poor..fair..good..**great**
Implementation?	none..poor..fair..good..**great**
Testing?	none..poor..fair..good..**great**
Cycles of ADIT?	**one**..few..several..frequent
Priority?	**no**..yes
Versions?	no..**yes**
Change Order Control?	**no**..yes
Internal Prototype?	**no**..yes
External Prototype?	**no**..yes
Alpha Release?	**no**..yes
Beta Release?	**no**..yes
Duration?	**short**
Effort?	**little**

Table 7.21: Survey: A Sequential Software Engineering Methodology: Part 1

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	**never**..seldom..often..frequent
Did an architectural change ever negatively affect accomplished work?	**never**..seldom..often..frequent
Are new features introduced up to product release?	**never**..seldom..often..frequent
Is there a dedicated period of quality assurance before the product is released?	no..**yes**

Table 7.22: Survey: A Sequential Software Engineering Methodology: Part 2

7.4.2 The Boehm-Waterfall Software Engineering Methodology

See tables 7.1 and 7.24 starting on page 153.

Question	Response
Analysis?	none..poor..fair..good..**great**
Design?	none..poor..fair..good..**great**
Implementation?	none..poor..fair..good..**great**
Testing?	none..poor..fair..good..**great**
Cycles of ADIT?	one..**few**..several..frequent
Priority?	**no**..yes
Versions?	**no**..yes
Change Order Control?	**no**..yes
Internal Prototype?	**no**..yes
External Prototype?	**no**..yes
Alpha Release?	**no**..yes
Beta Release?	**no**..yes
Duration?	**short**
Effort?	**little**

Table 7.23: Survey: Boehm-Waterfall : Part 1

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	**never**..seldom..often..frequent
Did an architectural change ever negatively affect accomplished work?	**never**..seldom..often..frequent
Are new features introduced up to product release?	**never**..seldom..often..frequent
Is there a dedicated period of quality assurance before the product is released?	no..**yes**

Table 7.24: Survey: Boehm-Waterfall: Part 2

7.4.3 A Cyclical Software Engineering Methodology

See tables 7.25 and 7.26 starting on page 154.

Question	Response
Analysis?	none..poor..fair..good..**great**
Design?	none..poor..fair..good..**great**
Implementation?	none..poor..fair..good..**great**
Testing?	none..poor..fair..good..**great**
Cycles of ADIT?	one..few..several..**frequent**
Priority?	yes..**no**
Versions?	**no**..yes
Change Order Control?	**no**..yes
Internal Prototype?	**no**..yes
External Prototype?	**no**..yes
Alpha Release?	**no**..yes
Beta Release?	**no**..yes
Duration?	**medium**
Effort?	**medium**

Table 7.25: Survey: Cyclical : Part 1

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..seldom..**often**..frequent
Did an architectural change ever negatively affect accomplished work?	never..seldom..**often**..frequent
Are new features introduced up to product release?	never..seldom..**often**..frequent
Is there a dedicated period of quality assurance before the product is released?	**no**..yes

Table 7.26: Survey: Cyclical : Part 2

7.4.4 The Boehm-Spiral Software Engineering Methodology

See tables 7.27 and 7.28 starting on page 156.

Question	Response
Analysis?	none..poor..fair..good..**great**
Design?	none..poor..fair..good..**great**
Implementation?	none..poor..fair..good..**great**
Testing?	none..poor..fair..good..**great**
Cycles of ADIT?	one..**few**..several..frequent
Priority?	**no**..yes
Versions?	**no**..yes
Change Order Control?	**no**..yes
Internal Prototype?	no..**yes**
External Prototype?	no..**yes**
Alpha Release?	no..**yes**
Beta Release?	no..**yes**
Duration?	**medium**
Effort?	**medium**

Table 7.27: Survey: Boehm-Spiral: Part 1

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	never..seldom..**often**..frequent
Did an architectural change ever negatively affect accomplished work?	never..seldom..**often**..frequent
Are new features introduced up to product release?	never..seldom..**often**..frequent
Is there a dedicated period of quality assurance before the product is released?	no..**yes**

Table 7.28: Survey: Boehm-Spiral: Part 2

7.4.5 The WaterSluice Software Engineering Methodology

See tables 7.29 and 7.30 starting on page 157.

Question	Response
Analysis?	none..poor..fair..good..**great**
Design?	none..poor..fair..good..**great**
Implementation?	none..poor..fair..good..**great**
Testing?	none..poor..fair..good..**great**
Cycles of ADIT?	one..few..several..**frequent**
Priority?	no..**yes**
Versions?	no..**yes**
Change Order Control?	no..**yes**
Internal Prototype?	no..**yes**
External Prototype?	no..**yes**
Alpha Release?	no..**yes**
Beta Release?	no..**yes**
Duration?	**many**
Effort?	**high**

Table 7.29: Survey: WaterSluice: Part 1

Question	Response
Did a new introduced requirement ever negatively affect accomplished work?	**never**..seldom..often..frequent
Did an architectural change ever negatively affect accomplished work?	**never**..seldom..often..frequent
Are new features introduced up to product release?	**never**..seldom..often..frequent
Is there a dedicated period of quality assurance before the product is released?	no..**yes**

Table 7.30: Survey: WaterSluice: Part 2

7.5 Summary

Here are some observations and thoughts from the surveys.

- Having a strong requirement process is a necessary condition for success but not a sufficient condition.
- A strong architecture that supports change is necessary for longevity.
- Implementation details, as long as the code is of high quality, are not a governing factor to long term life and success of a system.
- An overlooked requirement or architectural feature could lead to the downfall of a system.
- Having the right people is very important, but having critical features dependent on the performance of a few key individuals may lead to the downfall of the system.

Chapter 8

Conclusion, Future, and Related Work

The investigations that follow deal with software engineering elements that support change. These include methodologies (WaterSluice), paradigms (Noema), architecture (DADL), component composition (CHAIMS), and environments (DCE).

8.1 Methodologies

This thesis introduced the WaterSluice methodology. There are several potential follow on projects.

- Provide a tool suite to automate the WaterSluice methodology.
- Build tools in the WaterSluice methodology using a process definition language.
- Simulate the WaterSluice methodology.
- Tie the methodology to component engineering.

8.2 Paradigms

8.2.1 Abstract

A traditional engineering paradigm is very hierarchical in nature. To understand a whole, first understand the parts then combine the knowledge into an understanding of the whole.

In a noemic paradigm, the understanding of the whole comes first. The understanding of the part is a projection of the whole. The noemic paradigm reflects life.

This section [31] proposes that modern software engineering built for highly distributed computing environments should be based on a noemic paradigm.

Life is an example of a Noema. A Noema is not a neural network which simulates the learning process of the brain. A Noema is not a genetic algorithm which simulates system evolution. A noemic paradigm is represented by the body chemistry of living systems like the respiratory, circulatory, immune, and digestive systems.

This section will define the foundations of the noemic paradigm, give some examples, and support the conjecture that a Noema, though harder to build, supports change.

In a Noema, the whole is greater than the sum of the parts.

8.2.2 The Noemic Paradigm

Background

Traditional western science and technology is strongly influenced by rationalism and logical empiricism that can be traced back to Plato. A good summary of this paradigm can be found in [131]. When faced with the problem of trying to understand a system, the rationalistic tradition indicates that three basic steps are taken:

- Characterize the whole system in terms of identifiable sub-components with well defined properties.
- Understand each sub-component by finding general rules that describe their behavior.
- Combine the sub-components into the whole system, applying the rules of the sub-components, to draw conclusions about the behavior of the whole and to establish the understanding of the whole.

The rationalist approach requires complete knowledge of sub-components and their actions and interactions. Decomposition of complex systems into simpler parts is a natural scientific paradigm in the rationalist approach.

The rationalist approach is in contrast to hermeneutics [65]. Here the components of a whole system are defined as an interpretation in the context of the whole and the environment. There is no full and explicit understanding of neither the components nor the whole system. The understanding is never complete.

The whole system defines to exists a hermeneutic circle where there are no absolute facts but only interpretations of content within a context.

For example, try looking up a word in the dictionary. A word is defined in terms of other words which eventually have definitions which circle back to the original word. From Webster, the verb "to move" is defined as "to go from one place to another with a continuous motion" while the verb "to go" is defined as "to move on a course." Each word is defined in a circular fashion having each other's word used in each other's definition. The two words together form a noemic concept associated with motion. Of course, there are many meanings of these two words, each dependent on a context. These two words participate in many noemic concepts.

Hermeneutic circles are like fast spinning toy tops. An external observer, one outside the toy top, is given the tasks of riding, or understanding, the toy top. His first attempt is to step onto the toy top and is immediately thrown off. To be successful, first the observer must gain momentum, and match the motion of the toy top, and then, step onto the toy top. One can't understand the hermeneutic circle without first understanding the whole.

Edmund Husserl called the Hermeneutic circle paradigm a Noema [49]. Noema is an antiquated Greek word for an intellect.

A Noema has the following characteristics:

- The implicit beliefs within a Noema and assumptions cannot all be made explicit.
- Practical operational understanding of a Noema is more fundamental than detached theoretical understanding.
- A representation of a thing cannot be complete.
- Understanding is fundamentally in the context of the whole and cannot be reduced to activities of individual sub- components.
- A sub-component cannot avoid its interactions with the whole.
- The effects of the sub-components cannot be absolutely predicted.
- All representations of the current state are ephemeral at best.
- Every representation of a sub-component is an interpretation with respect to the whole.

- Every action of the sub-component affects the whole, even non action. The presence of the sub-component affects the whole.

The traditional hard sciences have carved a very small domain out of the universal Noema. The actions of one component affects the whole and cannot be taken in isolation.

Traditional Engineering

Traditional engineering is decompositional in nature. To understand the whole, first decompose the whole into the constituent parts. Then master the individual constituent parts, put them back together again, and the whole is now understood. The understanding of the whole is the sum of the understanding of the parts.

A good example of traditional engineering is a car. Break the car down into its parts, such as the steering sub-system, the transmission, the engine, the brakes, and many more sub-systems. The mastery of each of these sub-systems leads to the mastery of the car. The whole or, in this case, the car, is completely mastered by examining the parts in isolation and then looking at their combination to form the whole.

Noema

A Noema is different from traditional engineering. Any one part cannot be understood without the context of the whole. Changes in one aspect affect the whole. The role of a part is a projection into the whole. In traditional engineering, first understand the parts, then understand the whole. In a Noema, first understand the whole, then understand the role of each part.

A good example of a Noema is the human body. It contains the sub-systems of circulatory, digestion, nervous, and many others. But the role of each part is highly interdependent on the other parts. One often hears a physician say, "I need to get the total picture first before I can treat this patient." A change in one subsystem cannot be isolated from the other subsystems. Each individual cell acts independently, yet the whole is much greater than the sum of the parts.

Distributed computer environments are a Noema. This environment contains many highly interdependent components. Together they form a system.

Computer hardware is not a Noema. Hardware is based on hierarchical layering techniques appropriate for traditional engineering. A database is not a Noema, again for similar reasons. A life form is a good example of a Noema built over millions of years with natural selection and evolution. An information based economy is another example of a Noema.

My contention is that a Noema is the correct paradigm for the software engineering of large distributed systems.

8.3 Distributed Architectures

8.3.1 Abstract

Many computer science languages have been developed over the years that have concentrated on language fundamentals for the definition of algorithms and data structures. These traditional computer science languages give little help in defining the architecture of a system, especially a large distributed system. Architecture defines the components of a system and their interfaces, methods of communication, and behaviors. A Distributed Architecture Definition Language (DADL) [28] is proposed that extends the existing paradigm used in programming to include architecture descriptions for a particular class of distributed system architectures.

The architectural description language will provide fundamentals that concentrate on the conversation, communication, contracts, and behaviors of elements in the distributed system.

A DADL will be defined and used to describe a family of different distributive architectures. A program written in DADL can be compiled into different materializations of the architecture. Each materialization has different performance and resource characteristics leading to an optimizing choice.

It will be shown that large architectural variations can be described with minimal changes, thus showing the elaboration tolerance of DADL programs.

8.3.2 Introduction

Distributed Architecture Definition Languages (DADLs) are emerging as tools for formally representing the architecture of distributed systems. As architectures become a dominant theme in large distributed system development, methods for unambiguously specifying a distributed architecture will become indispensable.

An architecture represents the components of a large distributed software system and their interfaces, methods of communication, and behaviors. It is the behaviors of the components, the communication between the pieces and parts, that are underspecified in current approaches. To date, distributed system architectures have largely been represented by informal graphics in which the components, their properties, their interaction semantics and connections, and behaviors are hand-waved in only partially successful attempts to specify the architecture.

Traditional computer languages, like C, concentrate mainly on the definition of the algorithm and data structure components by using language provided mechanisms to specify type definitions, functions, and algorithm control. The interface is under-defined by header files where function names, parameters, parameter types, and parameter order are specified. This is short of specifying the behavior of the interface. Traditional computer languages are much more suited to defining implementation than they are to defining architecture.

Consider the following simple C program where we calculate the sum of two integers. See Table 8.1 on page 166.

The implementation file:

```
#include <plus.h>
void main() {
    int results ;
    results = plus(1,2) ;
} ;

int plus (int n , int m) {
    return n+m ;
} ;
```

The header file:

```
int plus (int n , int m) ;
```

Table 8.1: Example of a Traditional C Program with Header File.

Traditional programming languages easily define the data structures and the algorithms. There is very limited help in defining the architecture. In fact, there is an assumed architecture, so implicit that most languages don't even define it as a feature. The functions *main* and *plus* communicate over a shared address space, memory resident, ordered, highly reliable, synchronous, and error-free communication medium materialized by using a call-frame stack.

The language of communication is defined by the *call* statement. The function *main* sends two integers to the function *plus* and waits for an integer in reply. The function *plus* receives two integers and replies with their sum. The implicit *call* and *return* in the C language materialize this architecture.

This implicit architecture is appropriate for small and simple programs but as applications become more complex, large, and distributed, the implicit call-frame stack architecture is no longer appropriate. A distributed architecture might deal with a disjoint address space, non-memory resident, unordered, non-reliable, asynchronous, and error-prone communication mechanism. This is far from the assumptions of traditional computer languages. It is no wonder that large systems are hard to define using traditional programming languages.

Object based systems, like C++, extend the programming paradigm to include objects, sub-types, polymorphism, and inheritance. This powerfully extends the ability of a language to define the data structures and algorithms. However, the underlying implicit architecture does not change. The architecture still dictates memory resident, ordered, highly reliable, synchronous, and error free communication over a shared address space, that is materialized by a call-frame stack.

Another shortfall of the implicit object-based system architecture is in the definition of the behavior. Though the C++ interface defines the methods exported by a class, it does not define the methods used or required by that class. Thus an implementation can perfectly match the interface but have an entirely different behavior than another similar implementation because it is composed with different primitives.

Some of the founding object-based languages, such as SIMULA [77] and SmallTalk

[57], [85], [56], and [58], tried to replace the implicit architecture of a call-frame stack with a message-passing queue. In this architecture, methods are evoked by passing messages between objects. However, the architecture is still implicit and under-defined, leaving no choice in alternative behaviors.

Distributed middleware support systems, like DCE [54], extend the programming paradigm. The DCE Interface Definition Language (IDL) includes argument flow (in or out parameters), interface identifiers, dynamic binding information, and exceptions. Using DCE, it is possible to define communication mechanisms for architectures that are in disjoint address spaces, non-memory resident, non-reliable, and error prone. DCE accomplishes this by expanding the *call* mechanism. Asynchronous communication is dealt with by providing threads while unordered communication is provided by using network data grams under UDP. DCE replaces the traditional architecture with one that is more suited for distributed computing, but it does not allow a choice between alternative architectures.

CORBA [60] extends the programming paradigm to include messaging and distributed objects. Communication is done over an information bus where requests are issued and brokers respond to satisfy those requests. CORBA is really directed at building object models for a large class of applications under one, and only one, request/broker architecture. Though this is extremely necessary for application development, architectural needs go unfulfilled. CORBA is more like a detailed requirement specification, defining in detail the needs of a particular application domain.

Megaprogramming [130] extends the call mechanism to an asynchronous messaging paradigm between large components called megamodules. The communication between two megamodules is defined with language structures like setup, estimate, invoke, extract, and examine.

Languages, like Rapide [89], extend the interface definitions to include events and causal relationships between events. Using the paradigm of hardware design, the behavior of the interface is governed by signals and events which are synchronized by a clock. The interface has been extended to include both the generated and required methods. This allows for the interface to act more like a meta-schema that governs both actions and simple behavior.

In comparison, Rapide expands the role of the call statement into a directed graph of causal events. Megaprogramming expands the call statement into a family of asynchronous primitives. While the proposed DADL expands the call statement into conversations, behaviors, and contracts concentrating on distributed systems.

Don't confuse a DADL with a requirement language. The requirement is a statement of the problem at a high level of abstraction. This is in contrast to a DADL, which defines a generic plan that binds the requirements to the implementation. Requirement languages, such as STATEMATE [63] and Modechart [74], define the problem but not the solution.

This section proposes a DADL to specify architectures of distributed systems. This is accomplished by first defining the attributes of large distributed systems that distinguish a distributed system of other types of systems. Next, the DADL language will be defined. DADL will then be used to specify several key architectures.

Other related work includes Rapide [89], UniCon [120], ArTek [64], Wright [4], Code [97], Demeter [103], Modechart [74], PSDL/CAPS [90], Resolve [52] and Meta-H [126].

8.4 Component Engineering

The Compiling High-level Access Interfaces for Multi-site Software (CHAIMS) is a mega-programming language for software module composition [32]. The CHAIMS compiler is to generate a variety of invocation sequences for current and developing standards for software interoperation, with a focus on multi-computer, distributed operation. The language will include the ability to set up module interfaces prior to executions, request performance estimates from modules prior to their invocation, schedule modules in parallel, monitor execution of invoked modules, interrupt inadequately performing modules, and provide data and meta-information to customer interface modules.

CHAIMS supports a paradigm shift which is already occurring: a move from coding as the focus of programming to a focus on composition. This shift is occurring invisibly to many enterprises, since there is no clear boundary in moving from subroutine usage to remote service invocation. There are hence few tools and inadequate education to deal with this change.

8.5 Distributed Environments

Building a distributed application or infrastructure is tough. Many problems arise including security, communications, reliability, availability, serviceability, scalability and heterogeneity. OSF's Distributed Computing Environment helps solve many of these problems. See [30], [29], and [54].

Many organizations have distributed computing infrastructures where a large number of computers are connected together by a network. Powerful workstations are located in the offices of the employees serviced by even larger capacity servers.

Applications take advantage of the farm of computers by splitting the apparition into client/server partitions, where the graphical user interface resides on the workstation and the application rules and databases reside on the servers. These applications can communicate with each other and share information.

Many problems arise in distributed application engineering and systems. Some of these include communication, authentication, authorization, data integrity, data privacy, sharing of information, heterogeneous environments, distributed management, consistency of time, reliability, availability, parallel execution, and graceful degradation.

The Distributed Computing Environment (DCE) is a software component provided by the Open Systems Foundations (OSF) and supporting companies. Together, they have built solutions to the distributed application problems.

Appendix A

Software Life Cycle

A.1 Introduction

A system has a lifecycle consisting of many cycles from initial development, through deployment, operations, maintenance, legacy, and finally to discontinuation. The four fundamental phases of analysis, design, implementation, and testing can be applied to many different cycles and are not just limited to the development cycle as done in details in this thesis. See Figure A.1 on page 173 for a visual representation of the software engineering lifecycle.

The analysis phase establishes the goals. The design phase establishes the plan to accomplish the goals. The implementation phase builds the system, while the testing phase assures quality.

A brief sketch follows on the decomposition of other lifecycles into the phases of analysis, design, implementation, and testing.

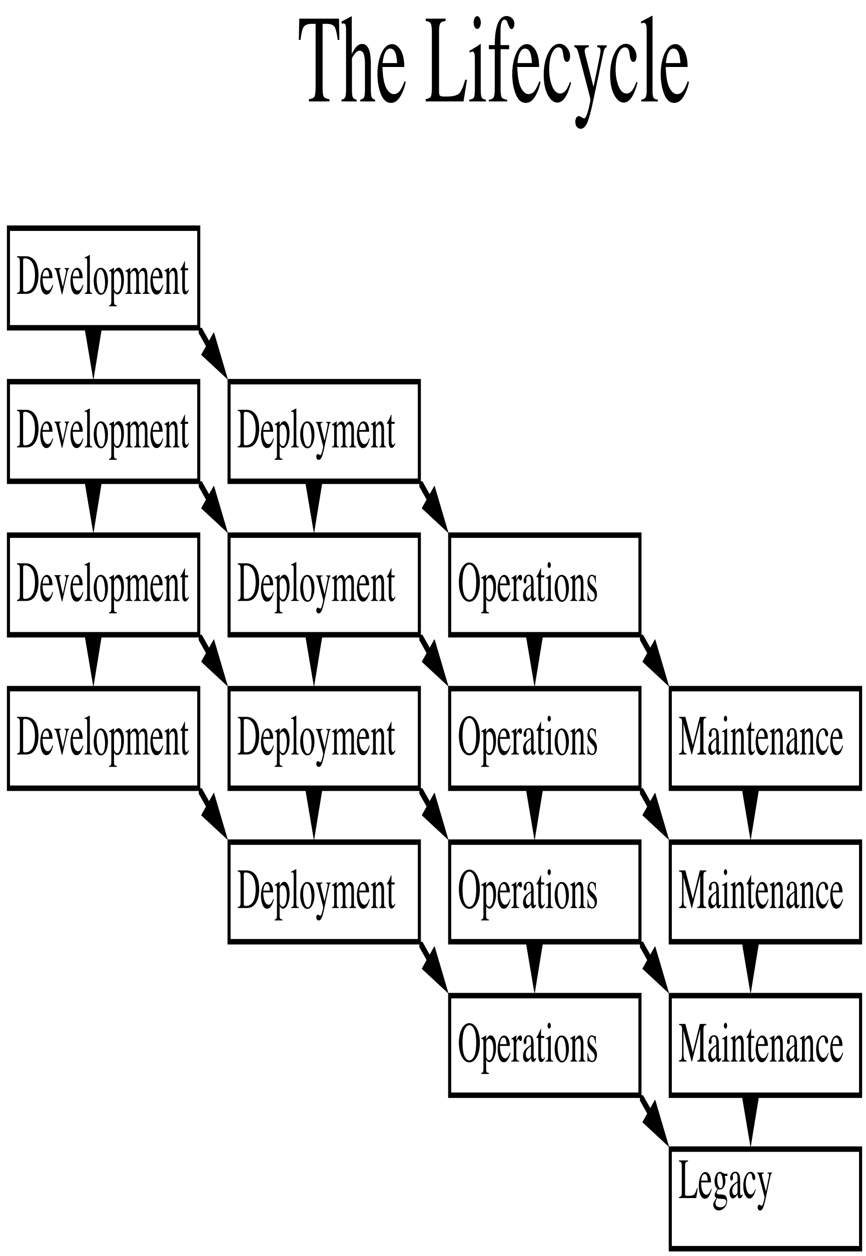


Figure A.1: The Software Engineering Life Cycle

A.2 Initial Development

Analysis

The analysis phase defines the requirements of the system.

Design

The design phase establishes the architecture of the system.

Implementation

The system is built in the implementation phase.

Testing

The testing phase improves the quality of the system.

A.2.1 GUI Development

Graphical User Interface (GUI) development is a special subcycle in the larger cycle of development of a system.

Analysis

The goal of GUI development is to establish an interface between the system and the customer that takes advantages of the powerful human-vision system.

Design

The architecture might include windows, icons, menus, sliders, check boxes, and a vast collection of other available widgets. A custom design that reflects current non-computer usage might be specified.

Implementation

In the implementation phase, the GUI is built.

Testing

Testing would include both improvements in the quality of the code and the usability of the system to the customer.

A.3 Version Deployment

Analysis

The goal of deployment is to make the system available for the customer.

Design

The design phase establishes a deployment plan and environment that will support the system.

Implementation

The plan is followed and the system deployed in the implementation phase.

Testing

Verify that the deployed system is operational.

A.4 Operations

Analysis

The goal of the operational cycle is to keep the system available to customer. This might include a goal of limited down time and 24 by 7 operations.

Design

The design might include a staffing plan, a version control system, a help desk, and a disaster recovery plan.

Implementation

The implementation phase is represented in the operations cycle by the availability of the system to the customer.

Testing

Reliability, Availability, and Serviceability (RAS) measures quality in the operational cycle. A system is reliable if it yields the same results on repeated trials. If a system is always available when the customer wants to accomplish a task, then the system is said to have high availability. If a system is fit for usage, it is said to have high serviceability.

A.5 Maintenance

Analysis

In the maintenance cycle, a change needs to be introduced to the system. This could be a new feature or a simple extension of an existing feature. The maintenance cycle may only provide corrections to discovered errors.

Design

The design phase in the maintenance cycle deals with how the change will be incorporated into the system.

Implementation

The implementation phase in the maintenance cycle deals with building the code that materializes the changes.

Testing

The regression test suite is modified to accommodated the change and is used to measure quality.

A.6 Legacy

Analysis

The goal of the legacy cycle is to freeze the system and to place the components of the system in escrow. No new changes are introduced, but the system needs to be available for limited customer usage.

Design

The plan might include the steps to escrow the hardware, the code, and the environments to run and build the system, possibly including the operating system, compilers, linkers, and databases.

Implementation

The implementation phase would carry out the steps necessary to bring the system to legacy status.

Testing

There is limited testing because there is limited change.

A.7 Final Discontinuation

Analysis

The goal would be to discontinue the system from all customer usage.

Design

The design would be the plan. This plan might reflect the building of replacement systems for discontinued services.

Implementation

The implementation would establish the replacements and turn off the system.

Testing

The testing phase would be limited.

Appendix B

The Supporting Engineering Environment

B.1 Introduction

Having a good methodology is important, but the methodology is only one piece of the whole solution. The engineering environment can be described in terms of groups.

Recall that there are four phases of analysis, design, implementation, and testing. A task is one action item from any one of the phases. A temporal arrangement of tasks is called a stage. The supporting engineering environment is established into groups.

Table B.1 on page 179 summarizes the supporting engineering environment.

Groups	Goals
People	Start with high quality people.
Tools	Give them powerful tools.
Strategies	Surround them with consistent directions.
Measurements	Measure their progress.
Feedback	Improve their productivity.

Table B.1: Supporting Engineering Environment

B.2 People

The first group is people. One of the most important parts of an engineering environment is to have the right people. If a project starts with the wrong people, then nothing else really matters.

Given a small project with a small number of really good people, little else is required. Good people will make things happen despite unforeseen difficulties. Unfortunately, having the right people does not scale for larger projects. As projects grow in people size, communication and coordination between people becomes the dominant controlling item.

B.3 Tools

The second group deals with tools. A person is only as good as the tools allow. The list of tools might include compilers, CASE, debuggers, DCE, CORBA, version control tools, project management tools, source control tools, life cycle tools, quality assurance and testing tools, database and transaction processing tools, bug and error report tracking systems, code test coverage tools, memory leak tools, change management, interface control, rapid prototyping, error and event management, event simulation, information sharing and file systems, security, and many, many more. Of course, two of the most important decisions are the operating system and the hardware.

B.4 Strategies

The third group deals with strategies. A person with a good tool needs to have direction and a strategy.

This includes methodologies which guide the generation of the requirement, architecture, and implementation plan. This includes the architecture of the system as well as points of view defined by the paradigms. A mission statement focuses the goals. There should be conventions on how to use the tools. Software system simulation and a test bed are a must.

There needs to be an understanding on how schedules, priorities, and decisions are made and established. The resource allocation algorithm needs to be defined. Task and skill definitions take place in this group, leading to potential staff training or changes.

Risk assessment is essential.

A list of strategies might include methodology, architecture, paradigms, mission, conventions, standards, schedules, priorities, decision process, resource management, risk management, and life cycle phases.

This section of the thesis deals with methodology. Other sections deal with architectures and paradigms.

B.5 Measurements

It is impossible to control what can't be measured. If we had no measurements, then no metric would show the effect of a change in a control parameter. The fourth group deals with collecting information about the environment. The measurements, results, and reports are all defined at this group.

A list of metrics might include the number of faults both reported and fixed, lines of code, closeness to plan, resource utilization, and performance. Lines of code is a problematic metric that does not work well in many situations, including composition. Many times the current progress is reported to be on plan, but when the plan indicates a deliverable, it is late.

B.6 Feedback

The last group is feedback. Plans can be monitored, leading to re-planning or plan repair. Ideally, plan optimization and the removal of chronic problems can be accomplished.

A list of feedback actions might include plan repair, re-planning, total quality management, continuous quality management, process changes, and plan optimization.

Appendix C

Requirements Gathering

C.1 Introduction

This chapter addresses the issues of modeling a real-world customer's need using requirement gathering techniques, fundamental doctrines, and tools.

The processes in a software engineering methodology transform a real-world customer's need into a computer system. The computer system is at best a model of the real-world customer's need. A close match between the computer system's behavior and the real-world customer's need's behavior enables the model to predict and simulate.

This thesis recognizes the importance of establishing the computer system model but can only give insights and not scientific guidance.

For more information on these topics see [3], [59], [70], [91], [108], [45], [125], [127], [129], [9], [34], [35], [55], [69], [133], [111], and [72].

C.2 Models

The distinction between declarative and imperative knowledge is well argued in computer science circles, especially in the field of artificial intelligence. Declarative knowledge represents the "what" knowledge, while imperative knowledge represents the "how" knowledge. Both are needed to gain full understanding.

A model is a representation of a real-world system. A model may use simplifying assumptions and approximations to capture only a portion of the real-world system. Given a sequence of inputs, the model makes a prediction. If these predictions match the real-world's systems reactions to the corresponding inputs, then the model is validated.

The customer has real-world needs. Requirement gathering generates a declarative model of the customer's needs. The architecture and implementation phases transform the declarative model into an imperative model. The testing phase assures that the generated imperative model matches and predicts the customer's real-world needs as expressed in the requirement's declarative model.

See Figure C.1 on page 184 for the three basic models of a real-world customer needs. See Figure C.2 on page 185 for the correspondence between the three basic models and declarative and imperative knowledge. See Figure C.3 on page 186 for the validation role of quality assurance.

C.3 Quality Assurance

There is no clean line between declarative knowledge and imperative knowledge. Frequently, a mixture of both kinds of knowledge are present.

Requirements seldom contain purely one kind of knowledge. Rather, a mixture of

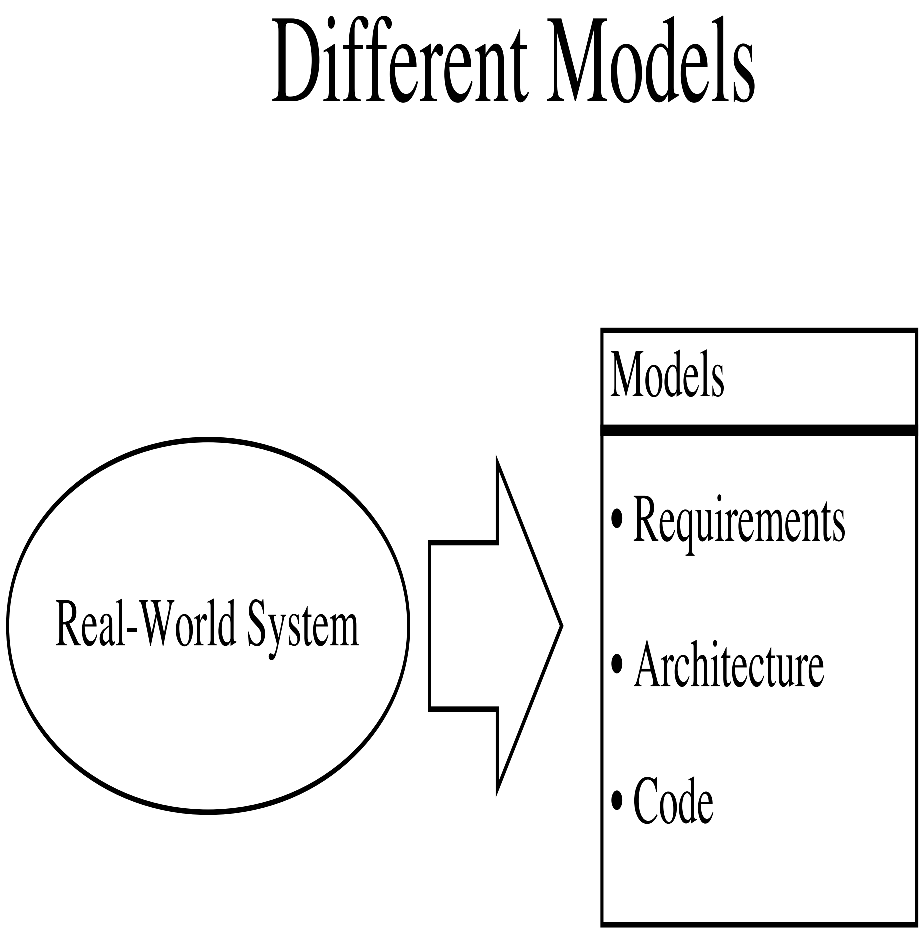


Figure C.1: Three Basic Models

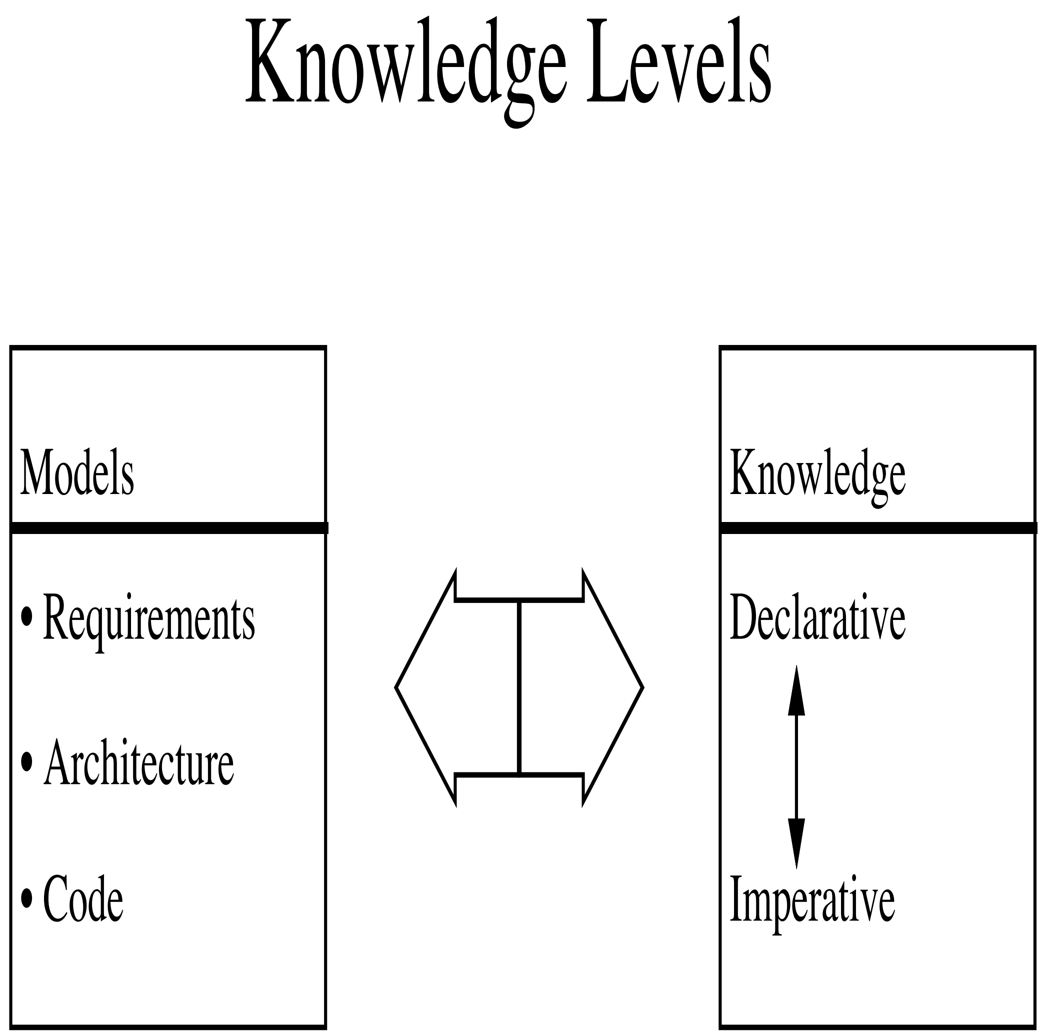


Figure C.2: Declarative and Imperative Knowledge

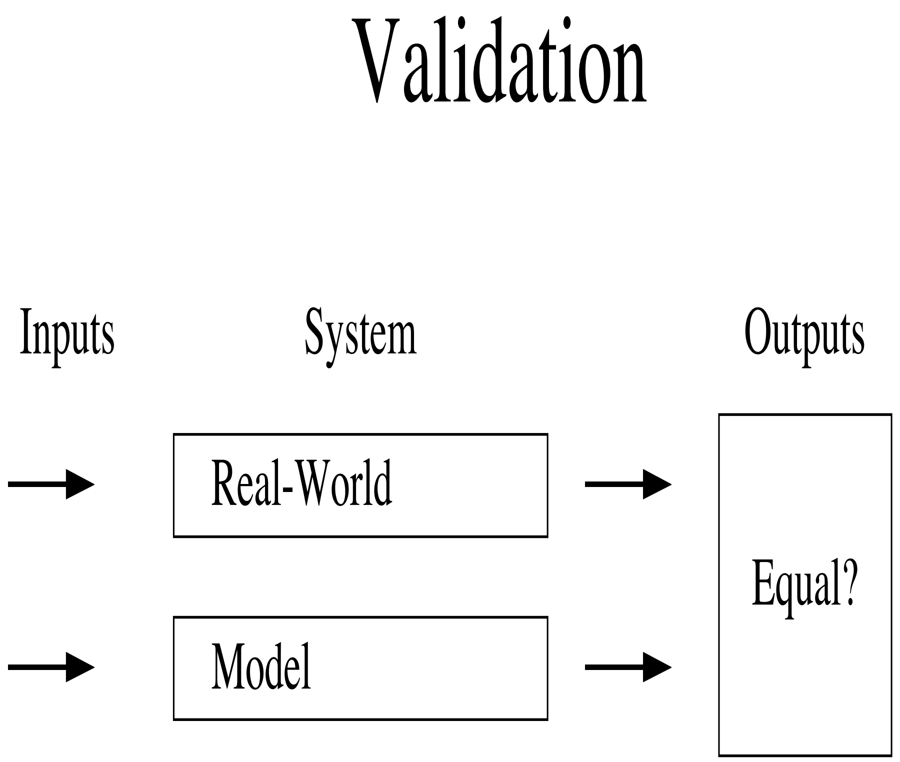


Figure C.3: Quality Assurance as Validation both kinds of knowledge are present. Some requirements are best modeled by declarative knowledge, while other requirements are best modeled by imperative knowledge.

Some requirement gathering techniques and tools emphasize one kind of knowledge over another. For example, The Fusion technique of D. Colemen et al defines the system operation schemes as declarative knowledge only, while the interaction graphs are defined as imperative knowledge only. This forced split becomes a hindrance for many real-world problems.

Another technique of Recursive-iterative development: 'Essays on Object-Oriented Software Engineering', Volume 1 by E.V.Berard, 1993, combines both kinds of knowledge at each level of abstraction. As one abstraction is refined into another abstraction, the combination of declarative and imperative knowledge is replaced with another combination of declarative and imperative knowledge.

The technique of Eiffel attaches declarative knowledge in the form of pre-conditions, post-conditions, and invariants to imperative knowledge. The imperative knowledge is materialized as algorithms and data structures.

Functional programming environments, like Mathematica, are declarative in nature. The system and not the customer discovers the imperative steps to solve the problem.

Mathematics also contains the two kinds of knowledge. The axioms and theorems are examples of declarative knowledge, while the proof is an example of imperative knowledge. Both are needed.

C.4 Storyboard

A storyboard is a technique used to rehearse typical customer scenarios before the application is built. It could consist of hand-drawn overheads used in a presentation to the team building the requirement document, including the customers. The storyboard conveys a rough idea of the application's behavior, the application's interaction with the customer, and visible components.

The typical scenarios of the system are rehearsed in this draft fashion. Often, significant errors can be caught and prevented before they propagate into other phases of the project. Alternatively, the storyboard might be a GUI-only version of the application where pictures of the output screens are presented. These GUI-only screens have no underlying application code but give an idea of what the customer will see and feel while running the application. The GUI builder in NextStep is a good example of a tool that generates high quality storyboards of an application complete with limited functionality and application stubs. An application stub contains only the interfaces and needs associated algorithms and data structures to complete the application functionality. In this case, if the storyboard is acceptable, then the application stubs can be implemented. See [122] and [6].

C.5 Some Fundamental Doctrines

The following levels of abstraction and points of views are taken from [36], [37], [39], and [38].

C.5.1 Abstraction

The process of establishing the decomposition of a problem into simpler and more understood primitives is basic to science and software engineering. This process has many underlying techniques of abstraction.

An abstraction is a model. The process of transforming one abstraction into a more detailed abstraction is called refinement. The new abstraction can be referred to as a refinement of the original one. Abstractions and their refinements typically do not coexist in the same system description. Precisely what is meant by a more detailed abstraction is not well defined. There needs to be support for substitutability of concepts from one abstraction to another. Composition occurs when two abstractions are used to define another higher abstraction. Decomposition occurs when an abstraction is split into smaller abstractions.

Information management is one of the goals of abstraction. Complex features of one abstraction are simplified into another abstraction. Good abstractions can be very useful while bad abstractions can be very harmful. A good abstraction leads to reusable components.

Information hiding distinguishes between public and private information. Only the essential information is made public while internal details are kept private. This simplifies interactions and localizes details and their operations into well defined units.

Abstraction, in traditional systems, naturally forms layers representing different levels of complexity. Each layer describes a solution. These layers are then mapped onto each other. In this way, high level abstractions are materialized by lower level abstractions until a simple realization can take place.

As Hoare [68] said,

> The major achievement of modern science is to demonstrate the links

> between phenomena at different levels of abstraction and generality, from quarks, particles, atoms and molecules right through to stars, galaxies, and (more conjecturally) the entire universe. On a less grand scale, the computer scientist has to establish such links in every implementation of higher level concepts in terms of lower. Such links are also formalized as equations or more general predicates, describing the relationships between observations made at different levels of abstraction.
>
> The general technique for crossing a level of abstraction is to define the way in which an observation at one level of abstraction corresponds to one or more observations at the other level. This relationship can itself be described by a predicate (often called a linking invariant) which relates an abstract observation (in the alphabet of the specification) to a more concrete observation (in the alphabet of the implementation).

See Figure C.4 on page 191.

Abstraction can be accomplished on functions, data, and processes. In functional abstraction, details of the algorithms to accomplish the function are not visible to the consumer of the function. The consumer of the function need to only know the correct calling convention and have trust in the accuracy of the functional results.

In data abstraction, details of the data container and the data elements may not be visible to the consumer of the data. The data container could represent a stack, a queue, a list, a tree, a graph, or many other similar data containers. The consumer of the data container is only concerned about correct behavior of the data container and not many of the internal details. Also, exact details of the data elements in the data container may not be visible to the consumer of the data element. An encrypted certificate is the ultimate example of an abstract data element. The certificate contains data that is encrypted with a key not know to the consumer. The consumer can use this certificate to be granted capabilities but can not view nor modify the contents of the certificate.

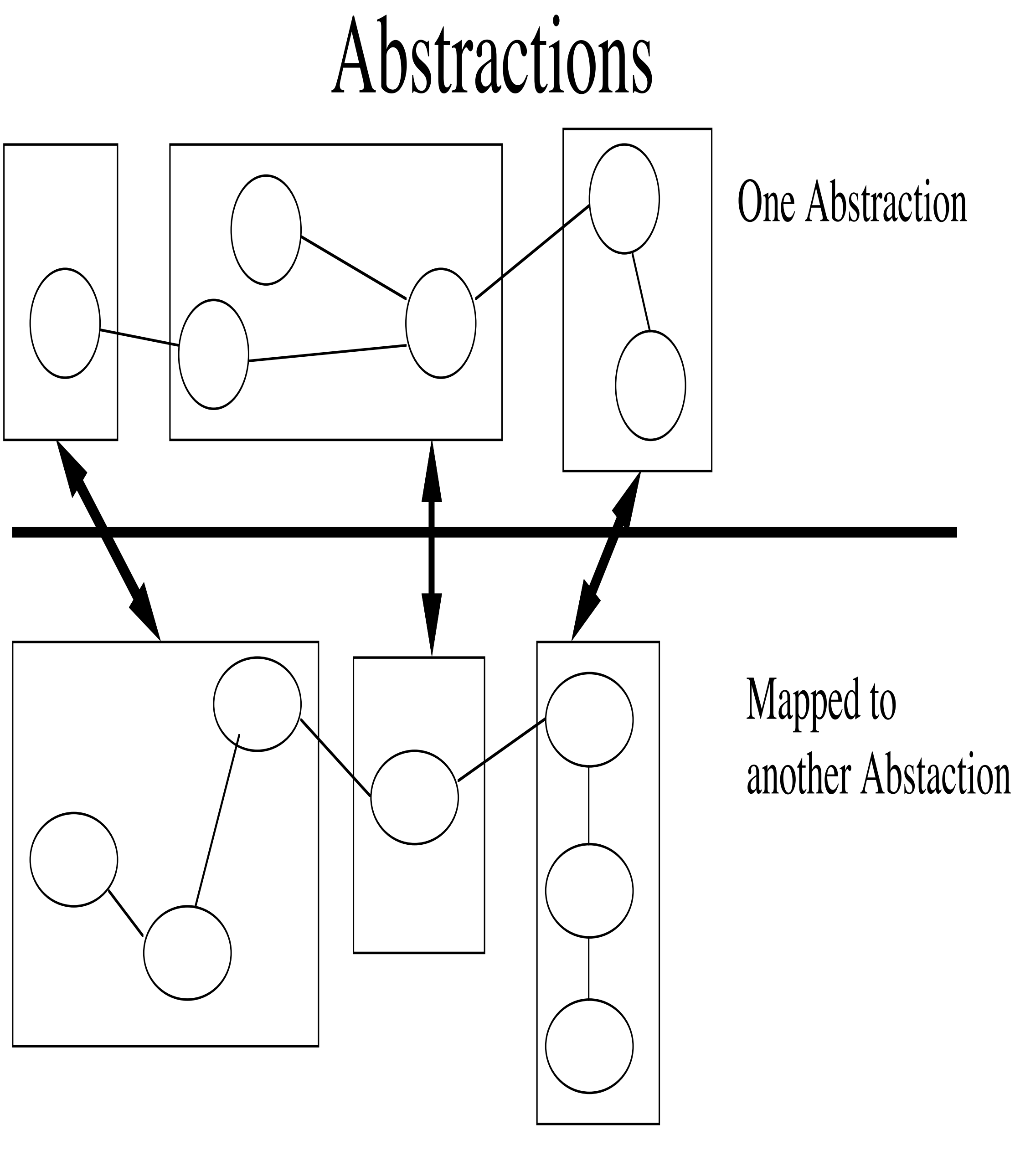


Figure C.4: Abstractions

Traditionally, data abstraction and functional abstraction combine into the concept of abstract data types (ADT). Combining an ADT with inheritance gives the essences of an object based paradigm.

In process abstraction, details of the threads of execution are not visible to the consumer of the process. An example of process abstraction is the concurrency scheduler in a database system. A database system can handle many concurrent queries. These queries are executed in a particular order, some in parallel while some sequential, such that the resulting database can not be distinguished from a database where all the queries are done in a sequential fashion. A consumer of a query which represents one thread of execution is only concerned about the validity of the query and not the process used by the database scheduler to accomplish the query.

C.5.2 Point of Views

A point of view is a way of looking at a problem. Each point of view generates a view. Views represent different ways in which the solution can be presented. Each view describes a solution. These views coexist. One view is not layered on top of another view; rather one view is expressed in terms of another view.

One of the views becomes the foundation view representation and the basis for all other views. The other views are then expressed in terms of the foundation view. See Figure C.5 on page 193.

For example, consider a relational database. There is one physical table architecture. With the aid of SQL you can express views of these tables. The view table is not physical but logical. The view table coexists with the other tables.

There is not just one view in a system, but many views. Each view describes the solution from a particular perspective. One view maps onto another. See [36], [37],

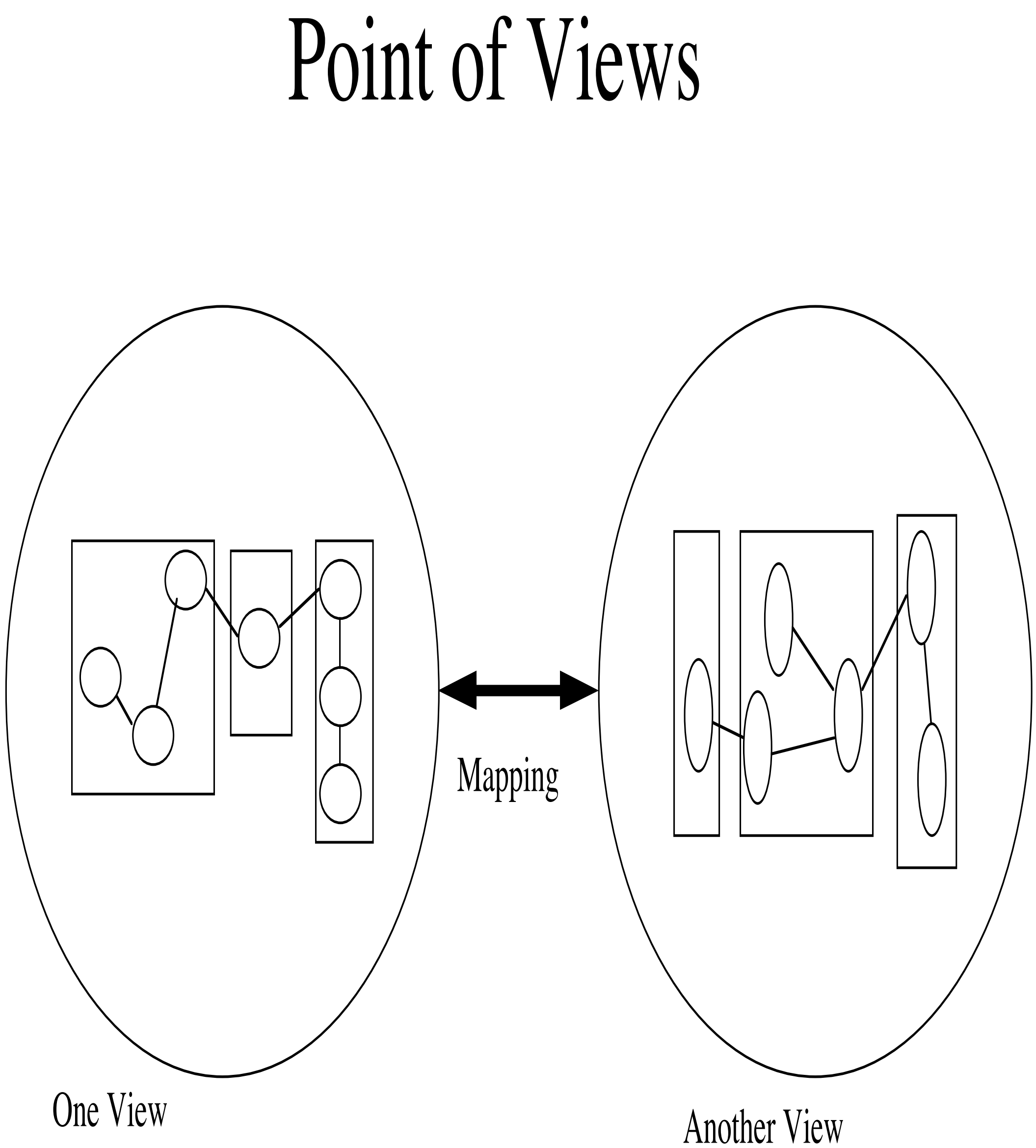


Figure C.5: Point of Views

[39], and [38].

Enterprise View

The enterprise view is used to capture and specify organizational requirements and structure. These are expressed in terms of policies, enterprise objects, communities, workflow, permissions, prohibitions, and obligations.

Information View

The information view is used to describe the data required. This is accomplished though the use of schemes, which describe the state and structures.

Some tools currently used to define the information level of abstraction include entity-relationship diagrams, conceptual schemes from OMT, and Z or other Formal Definition Techniques.

Computational View

The computational view defines the architecture. In this view, the pieces, the parts, interfaces, and behaviors are defined. Techniques such as encapsulation of data and processing, offered by the object-based paradigm, will be quite useful. There may be multiple interfaces and behaviors for any one part. The computational view is used to specify the functionality of the system.

There are three main components to the computational abstraction. They include the interface, the behavior, and the environment contract. The interface defines the types and functions. The interface also includes the well-formed sentences that are allowed. Some varieties of sentences could include a series of interrogations and announcements; a stream of non-atomic actions that continue throughout the lifetime of the interface; or signals and expected responding signals.

The behavior is described by a set of action sequences. The behavior may include some internal actions and is constrained by the environment. The environment contract includes quality of service constraints, real time constraints, usage and management constraints.

Infrastructure View

The infrastructure view defines the requirements for distribution and distribution transparency. The infrastructure view is also known as the engineering view. Each basic computational element corresponds to one or more basic distributed elements implemented with components of the infrastructure. Infrastructure components include the concept of a communication channel with marshaling and persistence, the concept of the hardware, the operating systems, the processes, and memory. A process may support deactivation, checkpointing, reactivation, recovery and migration.

Technology View

The technology view defines the hardware and software components of the implementation.

C.5.3 Scale

The model is at a particular scale. The more abstract and general, the higher the scale. The more concrete and detailed, the lower the scale. The scale might include differences in space units and in time units.

C.5.4 Classification

Classification is the systematic arrangement into groups or categories according to some established criteria. Classification establishes the taxonomy.

C.5.5 Generalization

Several classifications may only differ by a few concepts. Sometimes these concepts can be generalized into a more abstract representation; into a generalization. Many times, solutions are easier to establish in the more general case and then specifying a more detailed case.

C.5.6 Clustering

Sometimes several classifications may naturally be associated with each other, having many concepts in common. These classifications form a cluster. The clustering may be based on physical location.

In functional clustering, classifications are centered around functions. In data clustering, classifications are centered around data, while in object based clustering, classifications are centered around objects. The object based paradigm uses the concept of class hierarchies to naturally express clustering.

Encapsulation is a concept similar to the concept of clustering. In encapsulation, an enclosure is placed around a grouping and only threw well-defined openings are interactions allowed. A cluster may not have such a well defined enclosure.

C.5.7 Boundaries

Many interesting concepts have their most significant meaning at the boundaries. By exploring the boundaries, the true nature of the classification may be uncovered and understood.

C.5.8 Coupling

Coupling is the interdependence among individual components influenced by the interfaces, the kind of connections, and the kind of communications.

There are many different kinds of coupling. In data coupling, two components share the same data. Examples of data coupling include the usage of a data base to connect two systems. In stamp coupling, two components share a common data structure type which is passed in parameter calls. In common-environment coupling, two components communicate through a third party: the common-environment. In external coupling two components share a common-global data item.

C.5.9 Cohesion

Cohesion is the degree of closeness of the functions within a component. For example, consider a library of functions. If the library contains similar functions, then it is said to have high cohesion. A general purpose library, with many different functions, is said to have low cohesion.

C.5.10 Observations

Higher quality observations may lead to a higher quality model.

C.6 Components in a Requirement Document

A requirement document should contain the domain ontology, artifacts, actions, states, typical scenarios, and atypical scenarios.

See Section 2.2 on the analysis phase located on page 6.

C.7 Techniques

There are many different modeling techniques used to establish requirements. They include data models, process description, formal methods, textual specification, and use-case driven analysis.

Data structures and algorithm are fundamental in the field of computer science. Data modeling techniques concentrate on defining the data structures. Tools can be used to establish entity-relationship data models. Process description techniques concentrate on defining the algorithms. Tools can be used to establish flow charts.

The choice between these two techniques is driven by the task at hand. If the data represents a very stable concept in the model, then data modeling techniques will work best. If the processes represent a very stable concept in the model, then process description techniques work best. In my experience, data modeling techniques tend to be more fruitful than process description techniques.

A data modeling technique naturally leads to a database-centric system. A process description technique naturally leads to a computational-centric system.

Formal modeling techniques are declarative in nature. Formal modeling techniques may express the model in terms of mathematical logic predicates, usually based on first order logic. The trick is to establish tautologies that are always true under all interpretations and models.

Once the formal model is established, theorems in the model represent programs. A proof or deduction represents the algorithms and the structure of the model represents the data structure. A lemma represents a component and the corollary represents a reusable component.

A textual specification techniques establishes a model using a written natural language. Adjectives and nouns form noun phrases that represent data while verbs and adverbs form verb phases that represent algorithms or actions. The advantage of a textual specification technique is the expressiveness of language. The disadvantage is the ambiguity of language.

Use-case driven analysis techniques are centered around typical scenarios. This technique naturally leads to the definition of objects: a collection of both data structures and algorithms.

A similar technique of using storyboards leads to the definition of objects, states, and state transitions.

C.8 Summary

This chapter addressed the issues of modeling a real-world customer's need using requirement gathering techniques, fundamental doctrines, and tools.

Appendix D

Decision Making

Priority setting is only one of many decisions that are made on a project. A methodology does not state how a decision is made, just that a decision needs to be made. The WaterSluice methodology only requires that the priority decision process be goal directed to the final outcome of generating systems.

However, making decisions is so fundamental that a dedicated chapter is necessary to establish mathematical viability. See [50] for more details.

D.1 Alternative Tasks

A large project is divided into smaller alternative tasks. These alternative tasks may be independent or interdependent. Some alternative tasks may be inclusive while others exclusive. Picking one alternative task to accomplish next represents a choice and a decision point.

Let

$$T_1, T_2, \ldots, T_n$$

be the n alternative tasks.

D.2 Objectives

Objectives are used to measure the success of the system. Associated with each objective is a positive numerical weight indicating the weighted contribution of this objective to the final success of the system.

Let

$$O_1, O_2, \ldots, O_m$$

be the m objectives.

Let

$$W_1, W_2, \ldots, W_m$$

be the m weights.

D.3 Outcomes

An outcome is the estimated effect on the objective if an alternative task is accomplished. For every alternative task and for every objective is an outcome. The alternative tasks and objectives build the outcome matrix.

Let O be the outcome matrix

$$O = \begin{pmatrix} O_{11}O_{12}\dots O_{1m} \\ O_{21}O_{22}\dots O_{2m} \\ \vdots \\ O_{n1}O_{n2}\dots O_{nm} \end{pmatrix}$$

where O_{ij} is the outcome of the ith alternative task dealing with the jth objective.

D.4 Utility Function

A utility function transforms an outcome into a numerical value and measures the worth of an outcome. The utility of an outcome may be negative or positive. This utility function may be a simple table, a linear function, or a more complex function. The outcome matrix is converted to the utility matrix using the utility function.

Let U be the utility matrix

$$U = \begin{pmatrix} U_{11}U_{12}\dots U_{1m} \\ U_{21}U_{22}\dots U_{2m} \\ \vdots \\ U_{n1}U_{n2}\dots U_{nm} \end{pmatrix}$$

where U_{ij} is the utility of the ith alternative task dealing with the jth objective.

D.4.1 Temporal Utilities

The utility matrix may be a tensor over time. In this case the effective utility matrix is the temporal average.

If there are p time intervals then

$$U_{ij} = \sum_{q=1}^{p} (1/p)(U_{ijq})$$

D.4.2 Uncertain Utilities

The utility may be uncertain and risky. In this case, the effective utility matrix is a probability weighted average.

Let p_1 be the probability of an utility in state 1. Let p_2 be the probability of an utility in state 2 where $p_1 + p_2 = 1$

Then

$$U_{ij} = p_1(U_{ij})_1 + p_2(U_{ij})_2$$

is the effective utility.

D.5 Decision Rules

D.5.1 Weighted Sums

For each alternative task T_i calculate the weighted sum S_i.

$$S_i = \sum_{j=1}^{m} W_j U_{ij}$$

The alternative task associated with the highest weighted sum represents the decision.

D.5.2 Weighted Products

For each alternative task T_i calculate the weighted product S_i.

$$S_i = \prod_{j=1}^{m} {U_{ij}}^{W_j}$$

The alternative task associated with the highest weighted product represents the decision.

D.5.3 Deviation

For each alternative task T_i calculate the weighted norm S_i.

$$S_i = \left[\sum_{j=1}^{m} (W_j U_{ij})^2 \right]^{1/2}$$

The alternative task associated with the highest weighted norm represents the decision.

D.6 The Decision Process

The decision process is simple.

1. Establish objectives.
2. Establish alternative tasks.
3. Establish outcomes. For each alternative task and for each objective establish an outcome.
4. Establish utility. For each objective, establish the utility function and apply the utility function to every outcome. If the utilities have a temporal nature,

adjust to the effective utility. If the utilities have uncertainties, adjust to the effective utility.

5. Apply a decision rule.

Appendix E

Network Operating System

The requirement document for a Network Operating System (NOS) follows.

E.1 Introduction

A typical network consists of thousands of heterogeneous computers, each running their own operating system and managing their own peripheral devices and file systems. The computers share a common network with common services, but they are independent machines that have been extended to use the network. The customer can easily hop between machines. Logically, the network is thousands of independent machines hooked together with a common communication interface. To each machine, the network is just another peripheral communication device.

Now consider a network of thousands of heterogeneous computers, but under one network operating system (NOS). The network is the computer. All physical devices hanging onto the network, either CPUs or other peripherals, are network resources. Logically, the network is one computer that happens to have thousands of machines, peripherals, and services.

A customer authenticates to the NOS and is granted capabilities. A capability represents the permission to use a resource. The customer has control of all shared resources on the NOS, including a vast collection of services. Where the services are located is invisible to the customer. A customer's job may be run on a variety of available computers. From the customer's perspective, the desktop machine is transformed into a very powerful computer with a vast reservoir of resources.

Each desktop machine has a native operating system plus an extension that brings the native operating system into the NOS. This extension is a new network transport layer called principal to principal (PTP) that is layered on top of the existing TCP and UDP network layers, providing principal to principal communication. This layer provides a secure, authenticated, authorized, and private communication between two principals. A principal could be a person, a computer, or an application.

Along with the PTP transport layer is a NOS finder. The NOS finder is the boot application that knows how to find all other NOS applications.

A family of new protocols should be established that provide the underlying support for the NOS. These new protocols might include support for process management, virtual memory management, locks, events, transactions, and peripheral management to name a few. The NOS is message-based.

A desktop computer can be connected to the network in several ways. As a foreign desktop computer, the network looks like a traditional network providing basic network services and transport layers of communication. Nothing has changed from the more traditional view of a network.

As a secure desktop computer, only secure access to a gateway NOS machine is available. The secure desktop computer only needs encryption software to establish a secure link over traditional TCP communication. The secure desktop computer establishes a secure link to a known NOS machine. This known NOS machine now can act as a gateway for all other NOS services.

The NOS desktop computer is a peer member on the NOS. The NOS desktop computer requires a full installation of the PTP transport layer and the NOS finder.

There are a collection of core computers which provide the services of the NOS. A NOS consists of desktop computers under the control of the customer and a vast reservoir of resources provided by the core infrastructure NOS machines.

E.2 Goals

E.2.1 Simple

Simple to install, maintain, update, manage, and use.

E.2.2 High Availability

From the customer's perspective, the NOS should seldom be unavailable. The NOS could be slow and lose some services, but only infrequently be unavailable.

A system that has high availability is not as reliable as a system that has fault tolerance. Experience has shown that the cost-advantage tradeoffs favor high availability.

E.2.3 Support Change

As new technology becomes available, the NOS grows and bends to accommodate change.

E.2.4 Support Longevity

If something worked in the past, it should work now, though perhaps not with the best performance. It should take a long time for a service to be completely removed from the NOS.

E.2.5 Legacy Support

There should be an easy way to bring non-native NOS services into the NOS.

E.2.6 Local Machine Autonomy

Every local machine gets to determine the level of sharing with the NOS. Local operating systems and applications will continue to run even if the machine is a member of the NOS.

E.3 Components: Things and Actions

E.3.1 Universal Unique Identity (UUID)

Everything in the NOS has a UUID. The UUID is typed.

Actions on UUIDs include create, set type, get type, and check type.

E.3.2 Principal

People, machines, and services are principals. Every principal has a UUID, is authenticated, and is granted capabilities by the authorization service.

There is one special principal: NOS root. Each machine has an identity of local root.

Every principal has a password. For non-human principals, the password is stored in a local key table. Every principal has associated demographic information. Managing principals is the basis for account management.

Actions on principals include create, modify, delete, disable, enable, authenticate, and logout.

Actions on passwords include set and randomize.

Actions on key tables include create, modify, and delete.

E.3.3 Authentication

Given a principal's name and password, the NOS security service authenticates this principal using either public key or private key algorithms. Authentication is valid for a finite period of time. Proxy and delegation are supported.

A proxy is a principal that acts on another principal's behalf. For all practical purposes, the proxy becomes the principal. The security system allows the proxy to continue, but logs the fact that it was the proxy and not the principal that accomplished the task.

Delegation temporarily grants one principal's capability to another principal.

Actions include login and logout, delegate, and proxy.

E.3.4 Authorization

The authorization service grants capabilities to principals. Protected objects in the environment have access control lists (ACLs). Before access is granted to a protected object, the principal's capability is compared to the ACL and the associated permission is granted or denied.

ACL actions include create, modify, delete, and validate.

E.3.5 Data Privacy

Communication is protected with an encryption algorithm. Alternatively, objects may be encrypted. Digital signatures can verify the content.

Data privacy actions include encrypt, decrypt, sign, and verify.

E.3.6 Process Management

A process is a basic unit of execution. Each process has a principal identity and is authenticated and authorized. A process may be on many machines on the network and migrate to another machine for load balancing.

A process has a large sparse address space which is dynamically mapped.

Every process has a collection of ports used for communication. Each category of communication is supported by a message-based protocol. The ports use only secure, authenticated, and authorized communication. From the process view, the NOS is a collection of processes where all communication is done through ports. This includes process control, IO, exceptions, and faults.

Every process has a priority and at least one thread. A multi-threaded process may be running on a processor group.

Processes provide the services of the NOS and may form groups of equivalent services. The requesting customer gets one service from the group.

Process actions include create, delete, migrate, suspend, resume, and change priority.

Port actions include queue, dequeue, and wait.

Thread actions are similar to process actions.

E.3.7 Network Binary

One significant component of a process is an image. An image is similar to an application. The NOS associates an image with a process and then starts the execution. Images are built for particular hardware instruction set, thus dictating the class of machines that the image can run on. Some machines may have an instruction set emulator, which allows for some images to run on non-native machines. There is no universal network binary.

E.3.8 Distributed File System

The Distributed File System (DFS) provides for information sharing. Every principal has a home directory which is network mounted.

Normal actions supported by a traditional file system are supported, including record level access.

E.3.9 Disk Space Management

There are physical disk units under the control of the NOS.

Disk actions include seek, read, write, allocate, free, mount, and dismount.

E.3.10 NOS CPU Scheduling

A process may be on any machine in the NOS. Local scheduling is done by the local OS. The NOS will move a process from one machine to another in order to load balance the network. Of course, the cost of moving the process may be high, so the expected benefit should also be high.

Once a process is on a particular machine, the local operating system and scheduler takes over.

E.3.11 System Commands and Shell

There is a collection of control commands used to gather information and control the NOS. The command interface is programmed in a fashion similar to the UNIX shell.

System commands might include the UNIX equivalence of ps, jobs, cd, pwd, mkdir, rm, rmdir, date, time, clear, man, passwd, logout, lpr, lpq, lprm, ls, more, page, head, tail, mv, cp, file, chmod, chown, chgrp, <, |, >, &, >>, fg, bg, kill, echo, sleep, ctrl-z, ctrl-d, ln, tee, find, grep, nohup, wait, nice, renice, exit, set, setenv, pushd, dirs, popd, alias, uniq, sort, cmp, diff, tar, dump, restore, at, crontab, su, biff, compress, uncompress, crypt, tr, od, mount, unmount, whoami, tty, style, spell, awk, make, imake, sort, who, w, finger, talk, mesg, telnet, rn, and X.

Preferences could be stored in UNIX equivalent .login, .cshrc, .history, .plan, .project, .rhosts, .signature, .forward, and .vacation.

E.3.12 Run Time Library Support

The equivalence of a run time library now becomes a collection of run time services. Each service is a process running in least at one location in the NOS and potentially many locations. There is a namespace of services.

The path variable is a list of locations for finding services. When a service is requested, the path is walked to find the service. The elements in the path variable correspond to locations in the distributed file system.

E.3.13 Memory Management

The virtual memory system is now three-tiered. First, RAM on the local machine is viewed, then disk on the local machine is viewed backed by disk on the NOS.

NOS memory actions are used to manage movement of pages from the NOS disk to a local disk or local RAM.

E.3.14 I/O and Peripheral Device Management

Every peripheral device has a unique identity. A stub on the local machine understands the NOS I/O port protocol and translates the protocol to the series of local operating system calls to manage the local peripheral. The results are then returned back through the port.

Sample peripheral devices might include printers, CDROMs, disks, tapes, CD Recorders, and floppy disks.

Peripheral actions might include load or unload device driver and allocated or unallocated device. Some device specific actions might include read, write, and seek.

E.3.15 Networking

Needless to say, all computers in the NOS are network smart.

E.3.16 Time

Clocks are maintained in a consistent fashion across the NOS.

E.3.17 Transaction

There is a transaction manager to govern atomic actions. The transaction manager should support at least two-phased commit.

E.3.18 Distributed Locks

There is a distributed lock manager.

Lock actions include set, check, and release a read or write lock.

E.4 States

The NOS maintains state in the list of all processes, services, authenticated principals, peripherals, and machines.

Every principal has a family of states including authenticated, not authenticated, enabled, and disabled.

The NOS maintains a list of all available services and their physical locations.

E.5 Typical Scenarios

E.5.1 Day-to-Day

A customer authenticates to the NOS on an existing NOS smart machine. At the customer's disposal is the local machine and the NOS services. The Distributed File System contains the customer's home directory with the customer's preferences. Regardless of which machine the customer authenticates on, the customer's home directory with the customer's customizations are present. The customer can easily find a vast number of network resources. All resources appear local to the customer, but in reality, they are scattered throughout the network. The customer is very happy to find his favorite APL-based calculator still available for a quick matrix inversion. The customer logs off the system when finished.

E.5.2 Machine Setup

To bring a machine into the NOS, the customer first establishes the native local operating system. This might be Windows, NT, UNIX, Mac, Next, Mach, OS2, or any other existing operating system. The PTP transport is installed along with the NOS finder. The NOS finder is an application which can locate all other NOS applications. Using a boot program and the customer identity, the rest of the installation is completed.

E.5.3 Customer Setup

A customer will have to create a NOS principal identity.

E.5.4 Service Setup

The service is established in the service namespace defining the demographical information. The associated files are loaded into the DFS. If the service is NOS smart, no additional steps are needed.

If the service is NOS dumb, then the service needs to be wrapped into the NOS. This is accomplished by creating NOS peripheral stubs for the terminal, key board, mouse, hard disk, and any other needed peripherals. The NOS dumb service will be running on a NOS smart machine. When the service is activated, the peripheral stubs will send the information to the requesting customer machine.

E.5.5 NOS Setup

The NOS is a collection of core services. The network is established with the PTP transport layer. The core services include authentication, authorization, namespace management, DFS, process management, port management, virtual memory management, and peripheral device management.

E.5.6 Peripheral Device Setup

A peripheral device is installed onto a NOS smart computer. The device is given an identity in the namespace and a peripheral wrapper, making the device NOS smart.

E.5.7 NOS development

A developer works with a thread-smart language. The NOS development environment is similar to any other operating system development environment. An API family, now based on messages, ports, and protocols, is provided.

To create a NOS service is as easy as creating a run time library. The dynamic link process is replaced with a NOS smart finder.

E.6 Atypical Scenarios

E.6.1 Bring the NOS Down

Shut down all core services on the network. The remaining machines on the network still understand TCP or UDP but no other NOS network services are available.

E.6.2 Remove a Machine or Service

Remove the associated files and reboot.

Appendix F

Documentation

Table F.1 on page 219 summarizes the components of the generated customer manual and their origins.

Existing Document	Customer Manual
Requirements	Introduction Definitions of all components. Definitions of all actions. Definitions of all states.
Scenarios	How to accomplish ...?
Architecture	How does it work?
Implementation	Screen shots

Table F.1: The Customer Manual

The requirement documents, the usage scenarios, and the architecture documents have already been completed in the four phases. If documentation is accomplished during the four phases, the customer manual is easy to generate. Simply cut and paste sections of the already existing requirement and design documents along with screen shots from the application into a consistent customer guide.

Appendix G

Glossary

Analysis : The software engineering process that generates the requirements.

Architecture : The components of a system, their behavior and interaction.

Change Management : A software engineering process that controls changes to a system once a feature freeze has been invoked.

Change Order Control : When a component is completed, changes to the components are placed under this process to manage changes.

Component Testing : Testing major components of the system or the entire system with simple usage.

Critical Error : An error in the system that prevents the functioning of a usage scenario with no known work-around. Many features of the system may be working, but a critical error prevents the scenario from functioning under certain situations.

Critical Task : The element of a plan for building a system, upon which the success or failure of the rest of the system hinges. A system may have many critical tasks.

Design : The software engineering process that generates the architecture.

Feedback Level : The fifth level in a software engineering methodology support environment. Once measurements are available, parameters can be varied and their effects monitored.

Gold Standard : The expected and trusted results of the system against which all other results are compared. Current results that differ from the gold standard indicates an error condition.

Internal Testing : Testing at the lowest level of the system.

Implementation : The software engineering process that establishes the code for the system with the aid of people and tools.

Life Cycle : The cradle-to-grave existence of a software system from initial conceptions through development, deployment, version releases, and final phase out.

Measurement Level : The fourth level in a software engineering methodology support environment. This level deals with metrics. Measurement is the first step in gaining the ability to control.

Methodology : The body of methods, rules, postulates, procedures, and processes that are used to manage a software engineering project. An algorithm that finds a solution with a given performance in a given environment.

Ontology : The high level definitions of the objects, their actions, and behavior in a system. Sometimes objects are defined by their noun phrases and their actions defined by their verb phrases, while their behavior is defined by the communications.

Paradigms : A point of view about how to understand and solve a problem.

People Level : The first level in a software engineering methodology support environment. Having the right people on the project is very important.

Product : A system which is ready for general release to the market place.

Proof of Principle : When enough of the system is built that the developers can convince themselves that the rest of the development of the system can proceed, the system is said to be in a proof of principle state. This usually includes the successful implementation of the critical tasks.

Project Management : The software engineering process aided by tools which helps the expedition of a plan. Resources, priorities, tasks, schedules, and dependencies are coordinated.

Prototype : The system is ready for customer testing but not fully functional. There may be an early prototype called alpha and a later prototype called beta. Enough of the system is completed to convince customers that a real product will soon follow.

Regression Test : A testing technique where a suite of test cases are evaluated against the system to assure the expected behavior as established by the gold standard.

Requirements : The necessary capabilities and behavior of a system from the customer perspective defines the functional requirements. Non-functional requirements, for the most part, are introduced as an artifact of the design.

Risk Analysis : The software engineering process that defines the pro and con for each decision point and an estimate of probability for success and failure. Often a fall back position or an exit strategy may be defined.

Scenarios : A description of the typical and atypical usage of a system.

Screen Shots : A captured GUI window from a running system.

Source Control : The software engineering process aided by an application, defining the code versions and baselines of the system. The source control application also coordinates the changes done by a team of individuals.

Spiral : A cyclical software engineering methodology where the analysis, design, implementation, and testing phases follow each other in an iterative fashion as they spiral towards a solution.

State : A sequence of settings and values which distinguishes one time-space snap shot of a system from another.

Story Board : A software engineering process used to walk through a usage of a system, usually before major components are completed, to raise confidence in usability.

Stress Testing : Testing the system under a load which is higher than expected in actual usage.

Strategy Level : The third level in a software engineering methodology support environment. Given people with powerful tools, a direction is still needed as well as underlying foundations abput how to accomplish the task.

Testing : The process that assures a level of confidence in the quality of a product.

Test Plan : The sequence of steps needed to raise the level of confidence in the quality of the product.

Tools Level : The second level in a software engineering methodology support environment. High quality tools enhance the performance of the team and place a high water mark on the complexity of the systems which can be built.

Unit Testing : Testing related groups of functions.

Versions : A system can be defined as a queue of many stable versions. Potentially, each version reflects new functionality or improvements in quality.

Waterfall : A sequential software engineering methodology where the analysis, design, implementation, and testing phases proceed one after another, like water flowing over a fall.

WaterSluice : A best-first software engineering methodology where the analysis, design, implementation, and testing phases proceeded in a prioritized fashion, going after the gold nuggets first. As the method process proceeds, choices are constrained. The WaterSluice borrows the iterative nature of the spiral method along with the steady progression of the waterfall method.

Appendix H

Acronym Key

ACL	Access Control List
ADIT	Analysis Design Implementation Testing
CDROM	Compact Digital Read Only Memory
CHAIMS	Compiling High-level Access Interfaces for Multi-site Software
CPU	Central Processing Unit
DADL	Distributed Architecture Description Language
DCE	Distributed Computing Environment
DFS	Distributed File System
GUI	Graphical User Interface
GUIWIMP	Graphical User Interface, Windows, Icons, Mouse, and Pointer
IO	Input and Output
NOS	Network Operating System
NT	New Technology operating system from Microsoft
OS2	Operating System Two from IBM

OSF	Open Systems Foundations
PTP	Principal to Principal network layer
RAM	Random Access Memory
RAS	Reliability, Availability, and Serviceability
SQL	Sequal Query Language
TCP	Transport Control Protocol
UDP	User Datagram Protocol
UML	Universal Modeling Language
UNIX	AT&T trademark for THE Operating System
UUID	Universal Unique Identity

Bibliography

[1] J. R. Abrial. On constructing large software systems. In *Algorithms, Software, Architecture. Information Processing 92. IFIP 12th World Computer Congress*, volume A-12, pages 103–12, 7-11 September 1992.

[2] Hira Agrawal. Mining system tests to aid software maintenance. *IEEE Computer*, 31(7):64–73, July 1998.

[3] O. Al-Saadoon. Aura-cfg/e: An object-oriented approach for acquisition and decomposition of dfds from end users. *Seventh International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute*, pages 1–7, June 1995.

[4] R. Allen and D. Garlan. Beyond definition/use: Architectural interconnection. *Proceedings, Workshop on Interface Definition Language*, January 1994.

[5] American National Standards Institute. *Reference Manual for the Ada Programming Language*, February 1983. ANSI/MIL-STD 1815A. Also published by Springer-Verlag as LNCS 155.

[6] S. Andriole. Storyboard prototyping for requirements verification. *Large Scale Systems*, 12:231–247, 1987.

[7] Atria Software, Inc. Beyond version control: Evaluating software configuration management systems. Technical report, Atria Software, Inc., 24 Prime Park Way, Natick, Massachusetts 01760, February 1994.

[8] Stéphane Barbey, Didier Buchs, and Cécile Péraire. Overview and theory for unit testing of object-oriented software. In *Tagungsband "Qualitätsmanagement der objektorientierten Software-Entwicklung"*, pages 73–112, Basel, October 24 1996.

[9] Barros. Requirements elicitation and formalism through external design and object-oriented specification. *IEEE International Workshop on Software Specification and Design, Los Alamitos, California: IEEE Computer Society Press*, December 1993.

[10] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 1983.

[11] B. Beizer. *Software Testing Techniques, second edition.* Van Nostrand Reinhold, New York, 1990.

[12] Dorothea Beringer. The model architecture frame: Quality managment in a multi-method environment. In M. Ross, C. A. Brebbia, G. Staples, and J. Stapleton, editors, *SQM'95 Third International Conference on Software Quality Management*, volume 1, pages 469–480, Seville-Spain, 1995. Also available as Technical Report (EPFL-DI-LGL No 95/111).

[13] B. Boehm. Software engineering – as it is. In *Proceedings of the* 4^{th} *International Conference on Software Engineering*, pages 11–21. IEEE Computer Society Press, September 1979.

[14] B. Boehm. Software architectures: critical success factors and cost drivers. In *Proceedings of the* 16^{th} *International Conference on Software Engineering*, pages 365–365. IEEE Computer Society Press, May 1994.

[15] B. Boehm and R. Ross. Theory-w software project management: a case study. In *Proceedings of the* 10^{th} *International Conference on Software Engineering*, pages 30–40. IEEE Computer Society Press, April 1988.

[16] B. W. Boehm. Guidelines for verifying and validating software requirements and design specifications. *EURO IFIP79*, pages 711–719, 1979.

[17] B. W. Boehm. Software process management: Lessons learned from history. In *Proceedings of the* 9^{th} *International Conference on Software Engineering*, pages 296–298. IEEE Computer Society Press, March 1987.

[18] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the* 2^{nd} *International Conference on Software Engineering*, pages 592–605. IEEE Computer Society Press, October 1976.

[19] Barry W. Boehm. Software and its impact: a quantitative assessment. *Datamation*, pages 48–59, May 1973.

[20] Barry W. Boehm. Software engineering. *IEEE Transactions on Computers*, 100(25):1226–1241, 1976.

[21] Barry W. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, December 1976.

[22] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[23] Barry W. Boehm. *Software Risk Management.* IEEE Computer Society Press, 1989.

[24] B.W. Boehm. Using the winwin spiral model: A case study. *IEEE Computer*, 31(7):33–44, July 1998.

[25] B.W. Boehm, P. Bose, E. Horowitz, and M. J. Lee. Software requirements as negotiated win conditions. *Proceedings of ICRE*, pages 74–83, April 1994.

[26] Grady Booch. *Object Solutions.* Addison-Wesley, 1995.

[27] A.W. Brown, D.J. Carney, E.J. Morris, D.B. Smith, and P.F. Zarrella. *Principles of CASE Tool Integration.* Oxford University Press., New York, NY, 1994.

[28] Ron Burback. A distributed architecture definition language: a dadl. http://www-db.stanford.edu/˜burback/, 1997.

[29] Ron Burback. Distributed computing environment architecture. *DECORUM97*, page B12, March 1997.

[30] Ron Burback. *Distributed Computing Environment Lectures.* Stanford, 1997. ISBN 0-18-205-549-3.

[31] Ron Burback. An engineering paradigm: Noema. http://www-db.stanford.edu/˜burback/, 1997.

[32] Ron Burback, Louis Perrochon, and Gio Wiederhold. A compiler for composition: Chaims. *Fifth International Symposium on Assessment of Software Tools and Technologies (SAST'97)*, June 1997.

[33] Robert N. Charette. *Software Engineering Risk Analysis and Management.* McGraw-Hill, NY, 1989.

[34] P. Ciaccia. From formal requirements to formal design. *Seventh International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute*, pages 23–30, June 1995.

[35] P. Ciancarini. Engineering formal requirements: An analysis and testing method for z. *Annals of Software Engineering*, 1997.

[36] International Standards Committee. Information technology – basic reference model of open distributed processing – part 1: Overview. Technical Report ISO 10746-1/ITU-T X.901, International Organization for Standardization, May 1995.

[37] International Standards Committee. Odp reference model part 2: Foundations. Technical Report ITU-T X.902 — ISO/IEC 10746-2, International Organization for Standardization, May 1995.

[38] International Standards Committee. Odp reference model part 3: Architectural semantics. Technical Report ITU-T X.904 — ISO/IEC 10746-4, International Organization for Standardization, May 1995.

[39] International Standards Committee. Odp reference model part 3: Architecture. Technical Report ITU-T X.903 — ISO/IEC 10746-3, International Organization for Standardization, May 1995.

[40] Dan Conde. Bibliography on version control and configuration management. *ACM SIGSOFT Software Engineering Notes*, 11(3):81–84, July 1986.

[41] A. C. Coombes and J. A. McDermid. A tool for defining the architecture of Z specifications. In *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 77–92. Springer-Verlag, 1991.

[42] Trevor D. Crossman. Inspection teams, are they worth it? In *Proceedings of the 2nd National Symposium on EDP Quality Assurance*, Chicago, IL., March 1982.

[43] Siddhartha R. Dalal and Allen A. McIntosh. When to stop testing for large software systems with changing code. *IEEE Transactions on Software Engineering*, 20(4):318–323, April 1994.

[44] W. Decker and Jon Valett. Software management environment (SME) concepts and architecture. Technical Report SEL-89-103, NASA Goddard Space Flight Center, Greenbelt MD 20771, September 1992.

[45] M. Von der Beeck. Method integration and abstraction from detailed semantics to improve software quality. *International Workshop on Requirements Engineering: Foundations of Software Quality*, June 1994.

[46] Digital. *Vax/VMS Users Introduction.* Bedford, 1982.

[47] Digital. *Vax/VMS System Software Handbook.* Bedford, 1985.

[48] James H. Dobbins. *Handbook of Software Quality Assurance*, chapter Inspections as an Up-Front Quality Technique, pages 137–177. New York: Van Nostrand Reinhold, 1987.

[49] Hubert L. Dreyfus. *What Computers Can't Do: A Critique of Artificial Reason.* New York: Harper and Row, 1979.

[50] Allan Easton. *Complex Managerial Decisions Involving Multiple Objectives.* Wiley Press., New York, NY, 1973.

[51] Robert G. Ebenau. Inspecting for software quality. In *Proceedings of the Second National Symposium in EDP Quality Assurance*, 12611 Davon Drive, Silver Springs, MD 20904, 1981. DPMA Educational Foundation, U.S. Professional Development Institute, Inc.

[52] S. Edwards, W. Heym, T. Long, M. Sitarman, and B. Weide. Specifying components in resolve. *Software Engineering Notes*, 19(4), October 1994.

[53] Christer Fernström, Kjell-Håkan Närfelt, and Lennart Ohlsson. Software factory principles, architecture, and experiments. *IEEE Software*, 9:36–44, March 1992.

[54] Open Software Foundation and Open Group. *The Distributed Computing Environment*, 1996.

[55] M. Fraser. Formal and informal requirements specification languages: Bridging the gap. *IEEE Transactions on Software Engineering, 17, 5*, pages 454–466, May 1991.

[56] Goldberg and Adele. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984. ISBN 0-201-11372-4.

[57] Goldberg, Adele, and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983. ISBN 0-201-11371-6.

[58] Goldberg, Adele, and David Robson. *Smalltalk-80: The Language.* Addison-Wesley, 1989. ISBN 0-201-13688-0.

[59] L. Goldin and D. Berry. Abstfinder: A prototype abstraction finder for natural language text for use in requirements elicitation: Design, methodology and evaluation. *IEEE International Conference on Requirements Engineering, Los Alamitos, California:*, 1994.

[60] Object Management Group. *Common Object Request Broaker Architecture*, 1996.

[61] Dick Hamlet. Foundations of software testing: Dependability theory. *ACM SIGSOFT Software Engineering Notes*, 19(5):128–139, December 1994.

[62] Dick Hamlet. Software quality, software process, and software testing. In Marvin V. Zelkowitz, editor, *Advances in Computers, vol. 41*, pages 191–229. Academic Press, 1995.

[63] Harel, Lachover, Naamad, Pnueli, Politi, Sherman, and Shtul-Trauring. Statemate: a working environment for the development of complex reactive systems. *Proceedings of the 10th International Conference on Software Engineering, Singapore*, April 1988.

[64] Terry Hayes-Roth, Erman Coleman, and Devito Papanagopoulos. Overview of technowledge's dssa program. *ACM SIGSOFT Software Engineering Notes*, October 1994.

[65] Martin Heidegger. *What is Called Thinking?* New York: Harper and Row, 1968. Translated by Fred D. Wieck and J. Glenn Gray.

[66] B. Hetzel. *Program Test Methods*. Prentice-Hall, N.J., 1973.

[67] B. Hetzel. *The Complete Guide to Software Testing.* QED, Information Sciences, Wellesley, Mass., 1988.

[68] C. A. R. Hoare. Mathematical models for computing science. Copy is available at ftp.comlab.ox.ac.uk under directory /pub/Documents/techpapers/Tony.Hoare named mathmodl.ps.Z, August 1994.

[69] P. Hsia. A formal approach to scenario analysis. *IEEE Software, 11, 2*, March 1994.

[70] P. Hsia and A. Gupta. Incremental delivery using abstract data types and requirements clustering. *2nd IEEE International Conference on Systems Integration, Los Alamitos, California: IEEE Computer Society Press*, pages 137–150, June 1992.

[71] J. D. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Brueckner, O. Roubine, and B. A. Wichmann. Reference manual and rationale for the ada programming language. *ACM SIGPLAN Notices*, 14(6), June 1979.

[72] I. Jackobson. *Object-Oriented Software Engineering, A Use-Case Driven Approach.* Addison-Wesley, 1992.

[73] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992. ISBN 0-201-54435-0.

[74] F. Jahanian and A. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[75] W. D. Jones. Reliability models for very large software systems in industry. In *International Symposium on Software Reliability Engineering*, page 35, Austin, Texas, 1991. IEEE Computer Society Press, Los Alamitos, California.

[76] Stanley M. Sutton Jr. and Leon J. Osterweil. The design of a next-generation process language. Technical Report Technical Report 96-030, Department of Computer Science, University of Massachusetts at Amherst, January 1997.

[77] Nygaard K. and Dahl O.J. *The Development of the SIMULA Languages.* Norwegian Computing Centre (NCC) in Oslo, 1962.

[78] C. Kaner. *Testing Computer Software.* Tab Books, Blue Ridge Summit, Pa., 1988.

[79] C. Kaner, J. Falk, and H. Q. Nguyen. *Testing Computer Software.* Van Nostrand Reinhold, New York, 1993.

[80] L. J. Kenah and S. F. Bate. *Vax/VMS Internals and Data Structures.* Digital Press, Bedford, 1984.

[81] B. Kitchenham, editor. *Software Engineering for Large Software Systems.* Centre for Software Reliability, City University, Northampton Square, London EC1V OHB, UK, 1990. ISBN 1-85166-504-8.

[82] D. E. Knuth. *The Art of Computer Programming I: Fundemental Algorithms.* Addison–Wesley, Reading, Massachusetts, 1968.

[83] D. E. Knuth. *The Art of Computer Programming III: Sorting and Searching.* Addison–Wesley, Reading, Massachusetts, 1973.

[84] D. E. Knuth. *The Art of Computer Programming II: Seminumerical Algorithms.* Addison–Wesley, Reading, Massachusetts, second edition, 1981.

[85] Krasner and Glenn. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, 1983. ISBN 0-201-11669-3.

[86] Philippe Kruchten. A rational development process. *CrossTalk, 9 (7)*, pages 11–16, July 1996.

[87] Ronald Lange and Robert Schwanke. Software architecture analysis: A case study. In Peter H. Feiler, editor, *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 19–28, Trondheim, Norway, June 1991.

[88] Ytzhak Levendel. Reliability analysis of large software systems: Defect data modeling. *IEEE Transactions on Software Engineering*, 16(2):141–152, 1990.

[89] Luchham and Vera. An event-based architecture definition language. *to appear in IEEE Transactions on Software Engineering*, 1996.

[90] Luqi, Shing, Barnes, and Hudhes. Prototypeing hard real-time ada systems in a classroom environment. *Proceedings of the Seventh Annual ADA Software Engineering Education and Training (ASEET), Monterey*, January 1993.

[91] N. Maiden and A. Sutcliffe. Requirements critiquing using domain abstractions. *IEEE International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press*, pages 184–193, April 1994.

[92] Martin. Policy implications of medical information systems. Technical report, Office of Technology Assessment, Washinton, D.C. U.s. Government Printing Office, 1977.

[93] T. H. Martin and E. B. Parker. Designing for user acceptance of an interactive bibliographic search facility. *Interactive Bibliographic Search: The User/Computer Interface. Montvale NJ: IFIPS Press*, 1971.

[94] Tom Martin. *SPIRES*. PhD thesis, Department of Communications, Stanford University, June 1974.

[95] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[96] G. J. Meyers. *The Art of Software Testing*. John Wiley and Sons, New York, 1979.

[97] Newton and Browne. The code 2.0 graphical parallel programming language. *Proceedings, ACM International Conference on Super Computing*, July 1992.

[98] N. Nilsson. *Principles of Artificial Intelligence*, chapter 2. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1980.

[99] T. Olson, N. Reizer, and J. Over. A software process framework for the software engineering institute capability maturity model. Technical report, The Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[100] Leon J. Osterweil. Software processes are software too. In *Ninth International Conference on Software Engineering (ICSE'87)*, pages 2–13, Monterey, CA, March 1987.

[101] Leon J. Osterweil. Software processes are software too, revisited. In *Nineteenth International Conference on Software Engineering (ICSE'97)*, pages 540–548, Boston, MA, May 1997.

[102] M. A. Ould and C. Unwin. *Testing in Software Development.* Cambridge University Press, 1986.

[103] Palsberg, Xiao, and Lieberherr. Efficient implementation of adaptive software (summary of demeter theory). *Northeastern University, Boxton*, 10, January 1995.

[104] Edwin B. Parker. SPIRES, stanford physics information retrieval system. Technical report, Stanford University, Institute. for Communication Research, December 1967.

[105] Parnas. Why software jewels are rare. *COMPUTER: IEEE Computer*, 29, 1996.

[106] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11(3):259–266, 1985.

[107] Mark Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley, 1995. ISBN 0-201-54664-7.

[108] C. Potts and I. Hsi. Abstraction and context in requirements engineering: Toward a synthesis. *Annals of Software Engineering*, 1997.

[109] Veleria Quercia and Tom O'Reilly. *X-Window System User's Guide.* O'Reilly, Sebastopol, 1991.

[110] J. Rader, E.J. Morris, and A.W. Brown. Investigation into the state-of-the-practice of case integration. In *Software Engineering Environments. IEEE Computer Society*, pages 209–221, July 1993.

[111] B. Regnell, K. Kimbler, and A. Wesslen. Improving the use-case driven approach to requirements engineering. *Second IEEE International Symposium on Requirements Engineering*, 1995.

[112] RMS Ross Corporation, 44325 Yale Rd West, Chilliwack, B.C., Canada, V2R 4H2. *Ross Box System*, 1997.

[113] G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the* 16^{th} *International Conference on Software Engineering*, pages 201–210. IEEE Computer Society Press, May 1994.

[114] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proc. WESCON*, 1970.

[115] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the* 9^{th} *International Conference on Software Engineering*, pages 328–339. IEEE Computer Society Press, March 1987.

[116] Winston W. Royce. Managing the development of large software systems: Concepts and techniques. In *WESCON Technical Papers, v. 14*, pages A/1–1–A/1–9, Los Angeles, August 1970. WESCON. Reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, 1987, pp. 328–338.

[117] J. Rumbaugh, M. Blaha, W. Permerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[118] R. W. Scheifler and J. Gettys. *X Window System*. Digital Press, USA, 1990.

[119] G. Gordon Schulmeyer and James I. McManus. *Handbook of Software Quality Assurance*. Van Nostrand Reinhold, 1987.

[120] Shaw, Deline, Klein, Ross, Young, and Selesnik. Abstraction for software architectures and tools to support them. *Carnegie Mellon University*, February 1994.

[121] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[122] M. Shurtleff. Storyboarding and the human interfacel. *Software Development*, pages 55–56, July 1994.

[123] Mark G. Sobell. *A Practical Guide to UNIX System V*. The Benjamin/Cummings Series in Computer Science. Benjamin/Cummings, New York, N.Y., 2 edition, 1991.

[124] Douglas Troy. *UNIX Systems*. Computing Fundamentals Series. Addison-Wesley, New York, N.Y., 1990.

[125] I. Vessey and S. Conger. Requirements specification: Learning object, process, and data methodologies. *Communications of the ACM, 37, 5*, pages 102–113, May 1994.

[126] S. Vestal. Mode changes in a real-time architecture description language. *Proceedings, Proc. International Workshop on Configurable Distributed Systems: Honeywell Technology Center and the University of Maryland*, 1994.

[127] G. Weinberg. Just say no! improving the requirements process. *American Programmer*, October 1995.

[128] P. Westmacott. Process support environments and their application to large scale systems. In B. A. Kitchenham, editor, *Software Engineering for Large Software Systems*, London, 1989. Elsevier Science Publishers Limited.

[129] S. White. A pragmatic formal method for computer sstem definition. *Computer Science Ph.D. Dissertation, Polytechnic University*, 1987.

[130] G. Wiederhold, P. Wegner, and S. Ceri. Towards megaprogramming. *Communications of the ACM*, 35(11):89–99, 1992.

[131] Terry Winograd and Fernando Flores. *Undersanding Computers and Cognition: A New Foundation for Design.* Ablex Publishing Corporation, Norwood, New Jersey, 1985.

[132] Douglas A. Young. *The X Window System - Programming and Applications with X (OSF-Motif Edition).* Prentice Hall, Englewood Cliffs, 1990.

[133] L. Zorman. Requirements envisaging by utilizing scenarios (rebus). *University of Southern California Computer Science Department Ph.D. Dissertation, Los Angeles, California*, August 1995.

Index