INTEGRATION TESTING OF OBJECT-ORIENTED SOFTWARE

by

GORDON WILLIAM SKELTON

submitted in accordance with the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

PROMOTER: DR. A. L. STEENKAMP

JOINT PROMOTER: DR. C. D. BURGE

August 2000

****

Integration Testing of Object-Oriented Software

by

Gordon W. Skelton

Promoter: Prof A L Steenkamp

CoPromoter: Prof C D Burge

Abstract

This thesis examines integration testing of object-oriented software. The process of integrating and testing procedural programs is reviewed as foundation for testing object-oriented software. The complexity of object-oriented software is examined. The relationship of integration testing and the software development life cycle is presented. Scenarios are discussed which account for the introduction of defects into the software. The Unified Modeling Language (UML) is chosen for representing pre-implementation and post-implementation models of the software. A demonstration of the technique of using post-implementation models representing the logical and physical views as an aid in integration and system testing of the software is presented. The use of UML diagrams developed from the software is suggested as a technique for integration testing of object-oriented software. The need for automating the data collection and model building is recognized. The technique is integrated into the Revised Spiral Model for Object-Oriented Software Development developed by du Plessis and van der Walt.

Key Terms:

Software Testing; Integration Testing; Object-Oriented; Software Development Life Cycle; Process Model; Unified Modeling Language (UML); Software Quality; Software Quality Assurance; Software Modeling

Acknowledgments

I would like to thank the following persons:

Professor Steenkamp for all of her wisdom and advise. Her keen insight into the areas of software engineering and object-oriented development provided me with the framework within which I could place my research. Her friendship and willingness to help me at all times is greatly appreciated.

Dr. Burge who had the faith in me to start this whole process. His support in both my research and the time and effort required to finish this thesis helped through the laborious process which seemed to have no end.

I would like to thank both of them for the expenditure of their time and effort in helping me complete the task.

Janet Skelton, my wife, who put up with late nights and lost weekends. She put up with me spending endless hours researching and writing my thesis wondering if it would ever end.

List of Figures

CHAPTER 1

Context of the Research

CONTENTS

1.1 Introduction

Object-oriented software development is accepted as a valid process in the software development workplace. Over the past several years a number of different methods have been introduced for use in the development of object-oriented software, along with a collection of languages supporting the object-oriented paradigm: C++, Smalltalk, Ada, and Eiffel, to name a few. Each of these languages has been merged with various techniques for implementing object-oriented software. The major emphasis, during this period of time, has been on the analysis and design components of the software development life cycle and the actual implementation of object-oriented designs within the structure and syntax of specific languages. CASE tools have been developed that can assist the software developer in performing analysis and design, as well as, preserving the design in both a textual and graphical manner. Several of these tools provide for the linking of the object-oriented design to the actual coding of programs. One is able to

interactively design the user interface and data storage; however, it is the programmer's responsibility to complete the implementation. There has been less progress in the generation of procedural components. Still, one is now able to automatically generate code that supports classes.

As use of the object-oriented development model grew, software testing of object-oriented software did not receive equal attention. The complexity of object-oriented software requires that there be techniques and tools available that can aid the tester in understanding the language implementation, as well as, actually test the software. Due to this complexity, particularly for software written in C++, it is imperative that there exist a means by which the implementation can properly be verified against the object-oriented design. This thesis focuses on requirements for the development of a method for testing object-oriented software which overcomes some of the difficulties inherent in object-oriented software. These difficulties are actually a result of the benefits of object-oriented software development, namely abstraction, encapsulation, inheritance, and polymorphism. When one develops software using the object-oriented method and the C++ language one of the goals is to develop software that supports reuse. To create reusable software one turns to such concepts as generic programming and the use of pre-existing class libraries and templates. The more one uses these concepts, the more abstract and complex the final software becomes. This abstraction and complexity lead to increased difficulty in testing the software.

1.2 Problem Statement

Traditional integration testing examines the interactions among units as they are included in the builds. Modules are identified as subroutines or procedures. Attention is directed to the functional view of the implementation.

Object-oriented software, on the other hand, is composed of elements called classes. A class encapsulates the characteristics and behavior of this fundamental building block. Integration

testing of object-oriented software consists of testing the interaction between objects and groups of objects. These groupings are often referred to as components and conceptually are similar to subprograms in procedurally-based software.

Due to the fundamental differences in the construction of procedural and object-oriented software, it is necessary to reevaluate software integration and integration testing. Although there are CASE tools which generate code, symbolic debuggers and CASE tools do not adequately support or eliminate the need for integration testing. Chapter 5 discusses the strengths and weakness of Rational Rose, a CASE tool available for object-oriented software and development.

Traditional testing methods for integration testing of software developed were developed for procedurally-based software. Evaluation of these integration testing techniques must be made in light of the complexity of object-oriented software brought on by encapsulation, inheritance, and polymorphism. These characteristics of object-oriented software dictate that additional integration testing techniques be developed and evaluated.

Software integration, within the context of object-oriented development, requires that one have adequate knowledge of the objects involved in inter-object messaging, as well as, the overall functionality of the software under consideration. This knowledge is gained from the design and implementation views of the software and grows in detail as the software is produced.

1.3 Scope of the Research

The research presented in this thesis focuses on that portion of the software development life cycle concerned with integration testing of objects and subprograms. The object-oriented paradigm is examined in order to identify those elements of the paradigm that impact on integration and integration testing.

Integration testing is an important task in the software development life cycle. Examination of the currently proposed life cycle models provides insight into where integration testing lies. The proper location and timing for integration testing are identified and the software development life cycle is annotated to reflect integration testing's role. Attention is drawn to those areas within the life cycle which have an impact on the degree of integration testing required for object-oriented software.

To understand the relationship of object-oriented software and integration testing is the focus of this thesis. Traditional integration testing is examined in order to determine its contributions and limitations in regard to the testing of object-oriented software.

1.3.1 Hypothesis

The hypothesis is: *Increased knowledge of disparities between design and implementation assists in determining the cause of these disparities and eliminating defects in implementation normally isolated during integration testing.* In examining this hypothesis one is also concerned with the types of defects that can be isolated by the technique proposed in this thesis and how the presence of those defects will be identified.

1.3.2 Constraints on this Investigation

The research presented in this thesis is restricted to object-oriented software written in C++. C++ was chosen because of the overwhelming number of existing software systems written in that language, along with the fact that C++ supports the basic fundamentals of object-oriented software: encapsulation, polymorphism, and inheritance. It may be argued that the implementation of these object-oriented elements in C++ is less-than-satisfactory. However, it is not the purpose of this thesis to discuss the merits and weaknesses of different languages currently being used for object-oriented development. Recognition is given to the fact that many CASE tools support other object-oriented languages such as JAVA and Smalltalk, for example.

A prototype is developed to demonstrate the concept of using multiple views of the software for aiding in integration testing of object-oriented software. The prototype is limited to data processing software. Real time applications are outside of the scope of the research presented in this paper. Real time software requires that one be concerned with timing of the software. At present the technique does not address that area of software testing.

The technique developed in support of the research presented in this thesis is designed to analyze and test programs that have already been compiled and unit tested. All of the programs analyzed by the static analyzer are assumed to have been syntactically checked by a compiler and have compiled into executable code. The purpose of this technique is not to check the syntax of the code or to perform debugging tasks at the unit level.

1.3.3 Proposed Solution

Traditional integration testing techniques delay testing until the units are first integrated and continue until the software has been fully integrated and tested. System and acceptance testing follows integration testing. Comparison of the design and the implementation are not considered part of integration testing.

Linkage of the implementation with the design and analysis through the software development life cycle is used to provide insight into the software being developed. Taking that knowledge, an integration testing technique is developed.

The technique suggested in this thesis is based on the analysis and comparison of models representing the design and the implementation of the software under consideration. Information represented in the form of Unified Modeling Language (UML) diagrams, representing the design and implementation of object-oriented software, assists in the validation of the software design. Comparing these models of the system helps identify defects in the software traditionally found during integration testing. In addition, these UML diagrams assist in the development of test

cases used in integration testing by highlighting communication between components of the system. Components, for the purpose of this thesis are defined as either objects or subsystems.

1.4 Method of Investigation

To examine the effectiveness of using various models representing different views of object-oriented software for the purpose of integration testing, this thesis takes the approach of researching the foundation work on integration testing; examining the current state-of-the-art in testing of object-oriented programs; noting the important issues of software quality and its relationship to the software development life cycle; and presenting the Unified Modeling Language as a candidate for actually modeling the software. Once the basic research is complete and documented, software written in C++ will be used to test the effectiveness of the proposed technique.

Table 1.1 Steps in Investigation

1. Review Literature on Integration Testing of Object-Oriented Software
2. Review Procedurally-Based Integration Testing Techniques
3. Validate Weakness in Integration Testing Techniques for Object-Oriented Software
4. Determine Which Life Cycle Process is Most Appropriate for Object-Oriented Software Development
5. Focus on the Phase in the Life Cycle Process Where Integration Testing Occurs
6. Examine the Application of Different Views of Software to Integration Testing Based on the Use of these Views for Analysis and Design of Object-Oriented Software
7. Formulate a Conceptual Solution to the Problem of Integration Testing of Object-Oriented Software
8. Perform Experimentation by the Prototype Technique
9. Evaluate the Results

The investigation follows a logical flow from traditional integration testing to the development of a prototype for testing the new technique developed in this thesis. All pertinent aspects of

software development are included in the research in this thesis in order to build a framework in which to place integration testing within object-oriented software development.

Evaluation criteria are developed from evaluating the current integration testing techniques proposed by practitioners and researchers in the field of object-oriented software development. In addition, the proposed technique is evaluated in how it assists in the identification of integration defects in object-oriented software, in how it integrates into the software development life cycle and how it is scalable to larger software projects. These criteria, as outlined in Table 1.2, are intended to assist in recognizing the strengths and weaknesses of the technique, as well as, aiding in future research in the area of object-oriented software testing.

Table 1.2

Evaluation Criteria

1. Assistance in identifying presence of integration defects in object-oriented software
2. Relationship of technique to dynamic, event-driven aspects of object-oriented software
3. Integration of technique into the object-oriented software development life cycle
4. Scalability of technique to larger projects

Additional to these criteria, the strengths and weaknesses of the technique and its impact on software testing are also discussed.

1.5 Format of the Thesis

This thesis consists of seven chapters. **Chapter 1** provides an overview of the thesis along with a statement of the hypothesis and the delineation of the scope of the research and project implementation.

Chapter 2 examines the concepts of software quality and software quality assurance. The

relationship of the software development life cycle and software quality assurance is presented. Various software process models and frameworks are presented. Finally, the relationship of software testing and software quality assurance is examined.

Chapter 3 focuses on integration testing in general and on integration testing of procedurally-based software. A review of the literature on traditional testing is provided.

Chapter 4 expands this discussion on testing as it relates to object-oriented software. A specific section focuses on integration testing of object-oriented software.

Chapter 5 builds the case for the techniques presented in this thesis. Justification is made for the use of the different views and ties the software development life cycle in **Chapter 2** to the integration testing of object-oriented software.

Chapter 6 presents the actual tools and techniques used for testing C++ programs. The prototype developed for testing the hypothesis is discussed in this chapter.

Chapter 7 provides an evaluation of the results of using the technique presented in **Chapter 6.** The pros and cons of the technique are compared and results of the research are outlined. A summary of the research and of the demonstration presented in this paper is provided. Any need for additional research is also provided in this concluding chapter.

CHAPTER 2

Elements of Software Quality

CONTENTS

2.1 Introduction

Quality is an important concept in software development. Before one can begin to examine the relationship of quality assurance and testing, it is imperative that a comprehensive understanding of quality be established as it relates to object-oriented software development. The techniques and methods supporting these attributes are considered. Once the elements of software quality have been identified, the ways in which various software development methods assist in incorporating quality are examined. The role of testing and quality assurance is also discussed.

2.2 Software Quality

The IEEE Glossary of Software Engineering defines software quality as 'The degree to which software possesses a desired combination of attributes' [IEEE, 1990]. That statement does not give a very workable definition since it is not clear which attributes are important. Building upon Crosby's definition of quality, 'conformance to requirements', the meaning of software quality can be expanded to conformance with requirements [Crosby, 1980]. These requirements include the user requirements and the standards adopted by the software development group, as well as accepted standards such as those created by the IEEE and the ISO. Compliance with standards implies that the relevant standards of concern must be documented unambiguously. Testing must be carried out that measures these requirements and standards against the end product and vice versa. The end product consists of the requirements and its compliance with the presented standards. The issue of standards is discussed later in this chapter.

Verification, the process of measuring the implementation against the specifications or requirements, must be carried out during the entire life cycle and must include not only the software products but the process as well. The software development process itself should be viewed in the light of continuous process improvement, where the process is examined and changes are made that are aimed at improving the quality of the final product. Deming points out that it is often the process, not the worker, that ultimately affects the quality of a product [Walton, 1986]. Deming illustrates this point in his famous bead experiment in which individuals

are instructed to use a scoop to produce a product consisting of white beads. Within the raw material, beads, are a number of red ones reflecting flaws in the input materials. No matter how closely the workers follow the instructions they still produce an unacceptable product, red beads. Deming's point was that the process was flawed, not the desire or ability of the worker to produce quality (white) beads [Walton, 1986]. The same result can be derived from a software development process that is flawed.

Before the process can be properly evaluated one must have a working knowledge of what constitutes quality within software. Characteristics of software quality, as outlined by [Sanders and Curran, 1994] and presented in Table 2.1, may be used as guidelines for software development. When developing software, one should measure these elements of quality against the end product. Additional quality guidelines are required that relate to how the software is being developed, tested, and maintained. Guidelines in support of quality assurance and control should also form part of the methods that are used for software development and maintenance.

The following table is designed to provide an understanding of the elements of quality that correspond to quality software. Sub-elements of the primary quality concepts of functionality, reliability, usability, efficiency, maintainability, and portability are presented.

Table 2.1 Definition of Quality Characteristics

Functionality	
Suitability	Attribute of software that bears on the presence and appropriateness of a set of functions for a particular task.
Accuracy	Attributes of software that bear on the provision of right or agreed results or effects.
Interoperability	Attributes of software that bear on its ability to interact with specified systems.
Compliance	Attributes of software that make the software adhere to application-related standards.

Table 2.1 Contd.

Security	Attributes of software that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs and data.
Reliability	
Maturity	Attributes of software that bear on the frequency of failure by faults in the software.
Fault Tolerance	Attributes of software that bear on its ability to maintain a specified level of performance in cases of software faults or infringement of its specified interface.
Recoverability	Attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it.
Usability	
Understandability	Attributes of software that bear on the users' efforts for recognizing the logical concepts and its applicability.
Learnability	Attributes of software that bear on the users' effort for learning the application.
Operability	Attributes of software that bear on the users' effort for operation and operation control.
Efficiency	
Time Behavior	Attributes of software that bear on response and processing times and on throughput rates in performing its function.
Resource Behavior	Attributes of software that bear on the amount of resources used and the duration of such use in performing its functions.
Maintainability	
Analyzability	Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.
Changeability	Attributes of software that bear on the effort needed for modification, fault correction.
Stability	Attributes of software that bear on the risk of unexpected effect of

Table 2.1 Contd.

Testability	Attributes of software that bear on the effort needed for validating the modified software.
Portability	
Adaptability	Attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered.
Installability	Attributes of software that bear on the effort needed to install the software in a specified environment.
Conformance	Attributes of software that makes the software adhere to standards or conventions in a specified environment.
Replaceabililty	Attributes of software that bear on opportunity and effort of using it in place of specified other software in the environment of that software.

[Sanders and Curran, 1994]

[Thornton, 1992] uses quality elements and divides them into two major categories which represent technical quality and user quality. Table 2.2 presents another list of criteria for evaluating software quality. Some of the same quality elements are included, however, in this instance emphasis is placed on the issues related to the software development process and to standards which are applied to the development of the code. In addition, the elements of quality related to the external view of the end-user are included in this list of quality components. All of these items are quite important. These elements provide the software development group with a useful measure that can be applied to evaluating both the process and the product. This set of criteria is helpful in the development of quality software within a process that supports such development.

Table 2.2 Quality Assessment Criteria

Technical Quality Assessment Criteria	
A. Complexity	The complexity of the algorithms used, as well as the target of the system
B. Understandability	Ease of understandability of the system development's intentions
C. Maintainability	Ease with which the system can be corrected, adapted and/or enhanced over its lifetime
D. Reusability	Ability to make use of code or objects in other systems
E. Efficiency	How well the system makes use of the available resources
F. Testability	Ease of testing as must as possible of the functionality of the system in real life conditions
G. Portability	Ability to move a system across operating environments
H. Modifiability	Ability to change some of the functionality of a system
I. Consistency	Whether the design techniques and coding methods used are consistently applied across the entire system
J. Interoperability	Whether the system can communicate with other software packages, e.g., import or export data or launch other programs
User Quality Assessment Criteria	
A. User-friendliness	Ability of the system to be easily understood and used by human users
B. Response Time	Response time of the system must be acceptable
C. Reliability	Degree to which the system can be relied on to be available when needed and to produce correct information
D. Robustness	Degree to which the system is able to withstand incorrect usage or conditions without failure
E. Correctness	Extent to which a product satisfies its output specifications, independent of its use of computing resources, when operating under permitted conditions
F. Integrity	Avoidance of data corruption or loss

Table 2.2 Contd.

H. Security	Avoidance of unauthorized access
I. Flexibility	Ability of the system to satisfy different user requirements
J. Completeness	The system should satisfy all the user requirements, not just a subset of them
K. Utility	Extent to which a user's needs are met when a correct product is used

[Thornton, 1992]

Additional elements of quality must be devised that relate specifically to the software development process. The application of quality assurance and quality control are vital to producing quality software.

Quality focus can be either external or internal to the software itself. Internal elements of quality are dependent upon the skills of those individuals designing and developing the software. It is strongly suggested that one refer to the work of [Kernighan and Pike, 1999] and [McConnell, 1993] for further insight into those aspects of programming that contribute to overall software quality.

In addition to the limited discussion of quality and software development presented in this thesis, a number of other individuals have examined the issue of software quality and the elements that one should consider. Dunn, Glass, Weinberg, Deutsche and Willis, as well as others, have examined the issues surrounding software quality and quality assurance. The results of their research are important when considering what constitutes quality in software and how one manages the process in order to assure that such quality is built into the software product.

2.3 Software Process Models

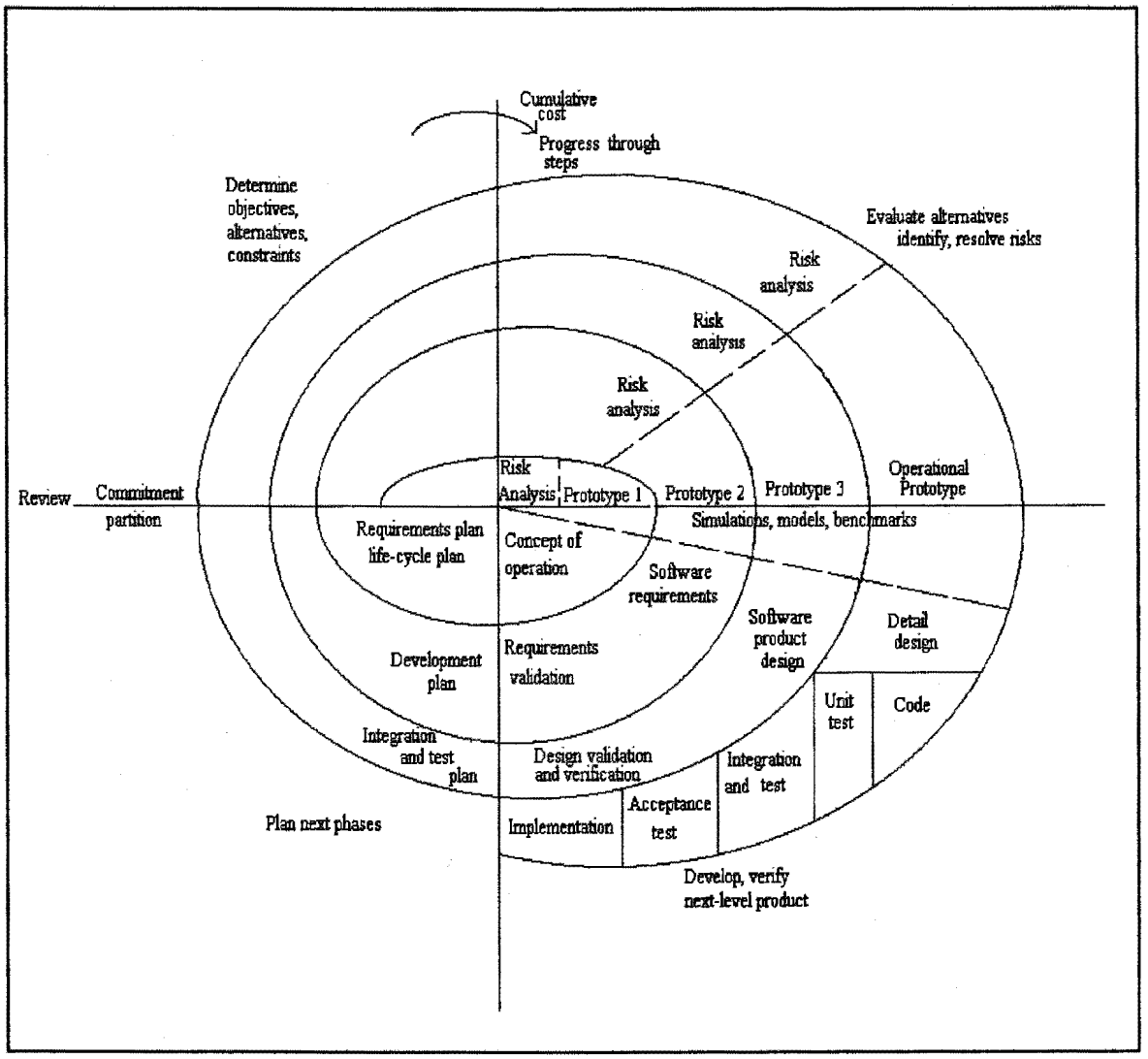
For software development to yield a quality product a suitable software process model must be used. Such a model formulates the transformation of a system's requirements to an executable software system by providing a framework within which the appropriate technical and managerial tasks may be applied. Various software process models have been proposed: the Waterfall Model [Royce, 1970], rapid prototyping, the incremental model, and the Spiral Model [Boehm, 1988], to name a few. Each of these models is discussed and compared in [Schach, 1999]. The Revised Spiral Model of van der Walt and du Plessis, presented in Section 2.3.1, was chosen as the life cycle model for object-oriented software to be associated with the integration testing technique developed in this thesis. The technique is considered to be an extension of their work on software development processes.

2.3.1 Revised Spiral Model (du Plessis and van der Walt, 1992)

[Boehm, 1988] developed the spiral model in an effort to include the best aspects of the classical method and prototyping, while at the same time resolving their weaknesses. He added risk analysis to the life cycle model which was not addressed by these models. The model, presented in Figure 2.1, contains a spiral that passes through four quadrants representing 1) planning, 2) risk analysis, 3) engineering, and 4) customer evaluation. The spiral depicts the iterative nature of software development while the four quadrants identify the four tasks that must be carried out during each development cycle. At the beginning of each cycle, risk is assessed and an evaluation of the project is performed. This evaluation determines whether the project proceeds as planned or if additional analysis must be carried out. The spiral model allows for changes in requirements throughout the life cycle.

As one can see from Boehm's spiral model, emphasis should be placed on the earlier elements of the software development process. Here the focus is on the analysis and design of the software. Decisions are made along the way as to the element of risk and whether or not the process should continue. There are major milestones where decisions are made regarding the continued commitment to the project.

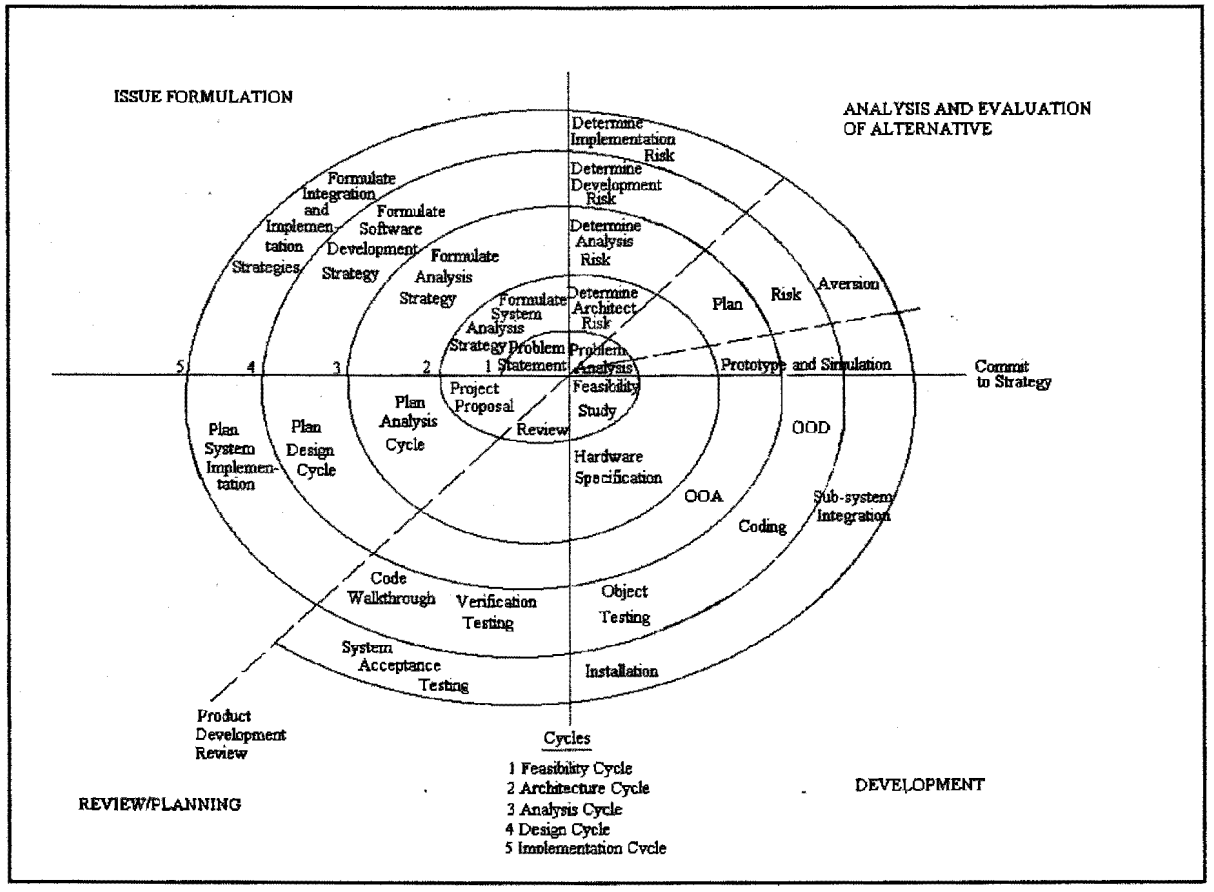
Boehm's spiral model incorporates the use of prototyping. As the process proceeds, prototypes are developed and evaluated until a point is reached where an operational prototype is developed. At that point the detailed design is completed and the actual coding, testing, and integration of the product are completed. System and acceptance testing is applied and the system is put into production.



[Boehm, 1988]

Figure 2.1 Spiral Development Model

[du Plessis and van der Walt, 1992] revised the Spiral Model for use with object-oriented software development. The Spiral Model was chosen because of the seamless development characteristics of object-oriented software development. Their model, presented in Figure 2.2, is composed of five cycles, the Feasibility Cycle, Architecture Cycle, Analysis Cycle, Design Cycle, and the Implementation Cycle.



[du Plessis and van der Walt, 1992]

Figure 2.2 Revised Spiral Model for Object-Oriented Software Development

Object-oriented analysis is performed during the analysis cycle. Here the object, dynamic, and

functional models are created. Object-oriented design is carried out during the Design Cycle. The output of this cycle is the detailed object, dynamic, and functional models, along with the subsystems. These subsystems are integrated during the Implementation Cycle.

Quality assurance is an activity that should be well integrated into the development life cycle. Starting at the very beginning of the process, criteria, as well as, function are established. The quality assurance function will be active throughout the life cycle, with particular importance during each of the assessments that occur at the end of each of the cycles.

Testing, which may be considered a sub-function of quality assurance, will also occur during each of the iterations. As discussed in Chapter 3 there are different types of tests which will be used during the life cycle. Beginning with unit testing, proceeding through integration testing, and finally culminating with system testing, testing will occur throughout the life cycle. Once the software has been developed and testing has occurred, end-users of the software will then perform acceptance testing where they will exercise the software, under either simulated or live conditions, and determine if the final product meets their needs.

2.4 Object-Oriented Development Methods

Building on the concepts that comprise the object-oriented model, a number of different methods have been proposed that contain techniques for object-oriented software development. Specific work has been carried out by Booch, Rumbaugh, Meyer, Jackson, and Jacobson, as well as others. Each of these individuals has contributed to the theorectical and practical foundations of the object-oriented method. The following sections discuss briefly the characteristics and concepts of the object-oriented paradigm, build upon the foundation work, and expand the discussion to recently presented methods. As with all process maturity, there is an evolutionary phase where newer process models replace older ones. This is true with the object-oriented method. To fully understand the object-oriented life cycle various methods are presented in this thesis. A number

of these methods have been suggested as replacements for the earlier ones.

The focus of the research reported in this thesis centers around the use of the iterative life cycle commonly associated with object-oriented software development. Included in that process is the need for attention on software testing and in particular integration testing. Special attention is placed on how a particular method relates to testing. Testing is an issue, however, that is not adequately treated by some of the methods or practitioners.

2.4.1 Booch's Life Cycle

The Booch Method [Booch, 1994] provides two different views of the system under analysis and design: 1) the logical/physical view and 2) the static/dynamic view. These views are required when one is specifying the structure and behavior of a system. Booch uses a number of different diagrams to support these different views: class diagrams, object diagrams, module diagrams, and process diagrams, along with two supportive diagrams: state transition diagrams and interaction diagrams. Booch defines these diagrams as:

- A class diagram is used to show the existence of classes and their relationships in the logical design of a system. A single class diagram represents a view of the class structure of a system.
- An object diagram is used to show the existence of objects and their relationships in the logical design of a system A single object diagram is typically used to represent a scenario.
- A module diagram is used to show the allocation of classes and objects to modules in the physical design of a system. A single module diagram represents a view of the module architecture of a system.
- A process diagram is used to show allocation of processes to processors in the physical design of a system. A single process diagram represents a view of the process architecture of a system.
- A state transition diagram is used to show the state space of an instance

of a given class, the events that cause a transition from one state to another, and the actions that result from a state change.

- An interaction diagram is used to trace the execution of a scenario in the same context as an object diagram.

[Booch, 1994]

These diagrams are used in the analysis and design phases of object-oriented software development.

Booch briefly mentions testing, stating that the three primary dimensions of testing, unit, subsystem, and system, should be carried out throughout the life cycle. Booch notes that testing should be directed toward the external behavior of a system with additional effort directed toward causing system failure under given circumstances.

2.4.2 Rumbaugh's OMT

Rumbaugh's Object-Modeling Technique (OMT) [Rumbaugh, 1991] relates to the software development life cycle as it handles the analysis, design, and implementation. Because OMT is language independent the implementation focus of OMT is restricted. However, the analysis and design phases are handled in great detail. OMT can be used with a number of different life cycle models. The Revised Spiral Model is well suited to the integration and usage of OMT since that life cycle is iterative which is well adapted to the object model. As the analysis and design phases proceed, objects are identified and the detailed object design of OMT is developed.

OMT provides three different views of the system under development: the object view, the dynamic view, and the functional view. The object model 'describes the static structure of the objects in a system and their relationships.' The dynamic model ' describes the aspects of the system that change over time.' Here the focus is on the internal control of the system. Thirdly, the functional model 'describes the data value transformations within a system.' OMT is divided

The Unified Modeling Language is employed in the analysis and design of software developed under the Unified Process.

2.4.4 Wirfs-Brock

Objects have responsibilities and collaborate with other objects in carrying out their responsibilities. Contracts are made between the objects so that it is clear which objects perform which tasks. Wirfs-Brock, as the method is referred to, is based on a life cycle software process model of two major phases: 1) the exploratory phase and 2) the analysis phase. The components of each are outlined in Table 2.3.

[Wirfs-Brock, et al., 1990] recognize the use of the spiral model for use in the development of object-oriented software. The object-oriented development life cycle is compared with the traditional, procedurally-based life cycle. For object-oriented software the emphasis should be placed more on design than on the implementation. Therefore, they recommend that the design phase of the life cycle should account for 50 percent of the time committed to the project. Without a good design it is virtually impossible to develop a software product that meets the requirements of the user, as well as, possesses the fundamental elements of quality.

Having discussed the analysis and design of object-oriented software, they spend limited time on the actual implementation of object-oriented systems. Consideration is given to the programming language of choice and how that language supports object-oriented elements such as inheritance and polymorphism. In choosing a language, it is helpful to have an integrated development environment that assists in execution, testing, and debugging of the system. With regard to testing, they believe that good design facilitates good testing. Individual classes are tested first and then the interfaces can be tested as the subclasses are included in the build. One should test all superclasses before testing any subclasses. First, one should validate that the subclass supports all of the contracts defined by its superclass then test any new contracts that the subclass introduces.

Wirfs-Brock relies on informal methods for the identification of objects. Designers rely on the requirements documents, as well as other input, to locate objects. This informal approach is both the strength and the weakness of Wirfs-Brock. This technique tends to produce a single model of a system. Under strict adherence to the method there is no dynamic view of the software.

Faults in the object-oriented design can be located because of the existence of individual entities. As entities are tested, they can be included into the system. Inconsistencies in component interfaces are easier to identify under the object-oriented model. Defects are likely to occur when component responsibilities are left out of the design or are assigned to the wrong entity.

Table 2.3 Wirfs-Brock Method

Exploratory Phase

Introductory

- read and understand specification
- utilize various scenarios during this phase to explore possibilities

Classes

- extract noun phases from the specification and build a list
- identify candidate classes from the noun phases
- identify candidates from abstract superclasses
- use categories to look for missing classes
- write a short statement for the purpose of each class

Responsibilities

- find responsibilities
- assign responsibilities to classes
- find additional responsibilities by looking at the relationship between classes

Analysis Phase

Hierarchies

- draw inheritance hierarchy graphs
- identify which classes are abstract/concrete
- draw Venn diagrams showing how responsibilities are shared between classes
- refine class hierarchy, checking the allocation of responsibilities
- define how responsibilities are clustered into contracts, and which classes support which contracts

Subsystems

- draw a complete collaborations graph for the system
- identify possible subsystems
- simplify the collaborations between and within subsystems

Protocols

- define protocols for each class by defining responsibilities into sets of method signatures
- write a specification for each class
- write a specification for each subclass
- write a specification for each contract

[Wirfs- Brock, et al., 1990]

2.4.5 Jacobson's Use Cases

Even though Jacobson's Use Cases cannot be classified as a true method, it does have an important impact on object-oriented analysis. Furthermore, Jacobson's Use Cases have become an important element in the UML.

[Jacobson, 1992] proposed the notion of use cases for exploring typical system behavior. A use case is a 'behaviorally related sequence of transactions which will be performed by the user in interaction with the system' [Jacobson, 1992]. Jacobson uses five views of the system for the development of his use cases: the domain view, the analysis view, the design view, the implementation view, and the testing view. For Jacobson use cases are very helpful in the analysis phase where one is attempting to understand how a particular system is to operate, specifically in how the user interacts with the system.

[Meyer, 1997], on the other hand, finds use cases a less-than-satisfactory means of identifying objects, and thereby classes. He identifies three primary reasons on should not rely on use cases for analysis and design. These reasons are: 1) use cases emphasize order, 2) relying on scenarios results in the designer relying upon the user's view of a system, and 3) use cases have a tendency toward functional analysis based upon specific actions. In each instance use cases result in violations of the object-oriented paradigm.

The benefit of use cases can be seen when employing them for validation of a new system under development. Use cases help evaluate a proposed system design, identifying missing components. Use cases aid in checking the system from a user's view point.

[Texel and Williams, 1997] focus their software development process on use cases throughout the life cycle. Building upon the work of [Jacobson, 1996], they employee use cases in the analysis, design, and testing phases. Use cases are merged with the spiral development model for viewing the software development process. Their primary focus is on the analysis and design

phases with additional, though limited, attention to software testing. Specific test cases and test drivers are developed for category, integration and system testing. In a separate phase [Texel and Williams, 1997] examine requirements tracing. Here their are interested in determining if the original requirements, identified in use cases, can be traced to the actual implementation.

2.4.6 Eiffel

Meyer's methodology presents a way in which one can handle the issues one faces when developing object-oriented software [Meyer, 1997]. To develop a system properly, Meyer states that one must have a good understanding of what the system is going to include and what it is not. Delineating the system helps in identifying what libraries to include for reuse and in the focus of the analysis.

First one must develop a list of the potential classes included in the system. Those classes are then grouped into clusters which are logical groups of classes. For Meyer the cluster is the unit for object-oriented development, not the class as some others consider and which is supported by this thesis.

System behavior is then modeled by developing object charts, event charts, and scenarios describing how the system behaves under given situations. External interfaces are developed which support the users' interaction with the system. Having completed these steps then the system is refined.

To handle all of these different phases, Meyer divides the system into clusters which can be developed concurrently. This parallel development eliminates the time delays which occur in traditional software development where work slows when one component is dependent upon the completion of another. Meyer's method still relies on the sequential process while allowing for iterative development within clusters.

The development life cycle of clusters reflects the traditional software development process of specification, design, implementation, verification and validation, and generalization. By using clusters one is provided with a means of having reversible development within the cluster or group of clusters without having to redesign the entire system under development.

Meyer's methodology is packaged in his language and process called Eiffel. Eiffel was created as a comprehensive approach to software development which includes all elements necessary for supporting the entire software development life cycle. Eiffel's key concepts are seamless development, reversibility, and contracting. Throughout the entire development process, one can use Eiffel for both the process and the programming language. This integrated approach is one of its attractions.

Eiffel, like Wirfs-Brock, employs contracts between components of which the system is comprised. These components, in order to build quality software, work with each other in a 'contractual' arrangement. Eiffel assures the proper working of the components through the use of class invariants, pre-conditions, and post-conditions. These quality enforcing concepts are built into the Eiffel language. Additional features of Eiffel assist the developer in testing the system. Eiffel allows the user to choose which assertions are to be monitored as the system executes.

2.4.7 The Catalysis Approach

The Catalysis Approach provides a process that incorporates a number of key object-oriented elements while at the same time following an iterative development life cycle. Emphasis is placed on the use of objects, components, and frameworks.

To implement a system under Catalysis the following steps should be followed: 1) specify, 2) document, 3) implement, 4) test, and 5) review. The process is considered to be nonlinear, iterative, and parallel in nature. [D'Souza and Wills, 1999] find the spiral development life cycle to work well in implementing their process in that it supports the five phases and is inherently

iterative. To illustrate the parallel aspect of their process they point out that the cycles can overlap and can occur concurrently. During the entire life cycle quality assurance and testing must be continually applied. To produce a final quality product, all intermediate deliverables must possess equivalent quality elements. The quality of the final product is the sum of the quality of the individual pieces.

When using Catalysis the degree of formalization in the specifications is dependent upon the likelihood of the component software will be used in the future. Code that is ad hoc and used once does not need the degree of detail in the design as code that has a longer half life or is to be reused in multiple systems. Even still, all specifications should be precise enough to support testing.

D'Souza and Wills recognize the importance of being able to trace from the specifications to the implementation. They point out one of the key claims of object-oriented development is traceability. The reasoning behind this is that "the classes in your program are the same in your business analysis so ... you should easily be able to see the effects of the design on any changes in the business." However, the reality of software development results in something quite different. As they note, "anyone who has done serious O-O design knows that in practice, the design gets pretty far from this simple ideal. Applying a variety of design patterns to generalize, coupling, and optimize performance, you separate the simple analysis concepts into a plethora of delegations, policies, factories, and plug-in pieces" [D'Souza and Wills, 1999]. Extrapolating on this idea, it is equally easy to see how one's implementation can be easily expanded so that the design is no longer traceable to the implementation.

The Catalysis process has three conceptual levels in the modeling of a system: 1) the domain model which illustrates the environment in which the software exists, 2) the component specifications which specify external behavior, and 3) component implementation which describes internal design. As reflective of the spiral development model, these three levels of models are recursively developed.

D'Souza and Wills develop a process that incorporates a number of key design ideas. The use of components and frameworks are integrated into both the design and the implementation. In order to facilitate their designs and to provide for a degree of traceability they utilize UML in order to have an unambiguous modeling language, one of their requirements for proper object-oriented design and development.

2.5 Quality Assurance and the Software Development Life Cycle

To fully comprehend the issues surrounding software quality and quality assurance, one must have a set of goals and standards which are included in every software system considered to possess quality. The elements included in quality software are presented in Section 2.2. To assure that these elements are properly implemented, there must be a set of standards used to guide the entire software development life cycle. In addition to these standards there must be an oversight function that manages and evaluates the development process in order to help assure that the resulting products possess quality.

Along with measuring the quality of software, the quality assurance (QA) function must also evaluate the development process, looking for areas where improvements can be made. As with other business processes there should be a commitment to continuous process improvement in software development. That improvement relates to both the process and the end products, along with the skill levels of the individuals involved in the development process.

The development of software includes both the process of manufacturing the software and the management of that process. In the management component, one must be involved with both evaluation of the actual product, to insure that the elements signifying quality are contained in the end product, and oversight, to assure that the development process is followed. Quality concepts must be built into the life cycle at all points. In doing so quality standards are employed that reflect both the individual organization's quality philosophy, as well as, the professional

community's accepted standards.

The following sections examine the concepts surrounding quality assurance and its relationship to the software development life cycle. Included in this discussion are standards and their application to quality assurance.

2.5.1 Role of Quality Assurance

To understand how quality assurance relates to the software development life cycle it is first imperative that one develops a working definition of quality as it relates to software. Section 2.2 discusses briefly those elements considered as important indicators of quality within a software product. Having established quality elements, it is then necessary that one create an operational framework within which quality products can be built. This framework relates to both the process by which software is developed and the quality assurance function which is responsible for measuring the quality of the software and evaluating the effectiveness of the process to produce quality software. The quality assurance function should be responsible for evaluation of both the process and the product, identifying weaknesses and flaws, and making recommendations for changes in action that bring the product back within quality parameters. Furthermore, the QA group should also determine how effectively the development process is being employed.

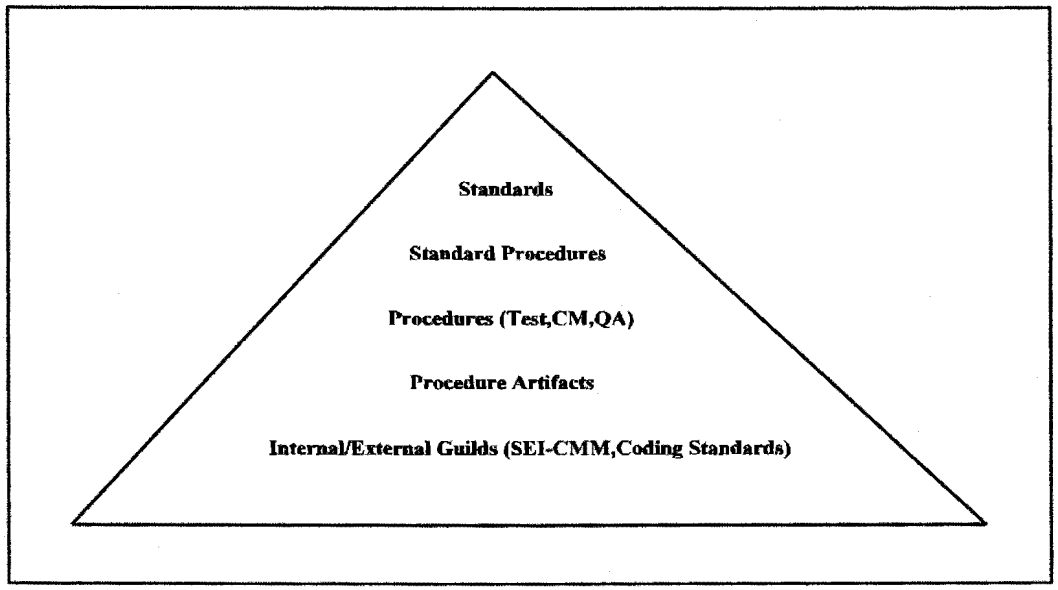
The Software Engineering Institute uses the following definition, derived from the IEEE-STD-610:

> Software quality assurance is:
>
> 1. A planned and systematic pattern of all actions necessary to provide adequate confidence that a software work product conforms to established technical requirements.
>
> 2. A set of activities designed to evaluate the process by which software work

products are developed and/or maintained.

[Paulk et al., 1995].

The definition includes the management of the software development life cycle in order to assist in and manage the development of quality software. The quality of a software product is directly related to the organization and the integration of the internal and external elements that support the process of developing quality software. Figure 2.3 illustrates the structure of a quality system.

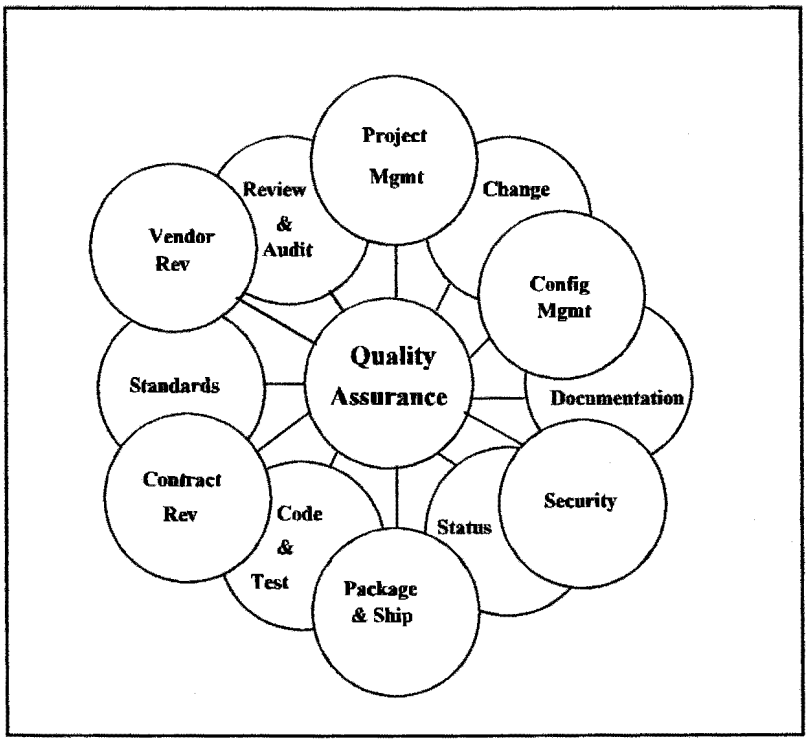


[Frank, et al., 1996]

Figure 2.3 Quality System Hierarchy

Quality assurance is the center of the software development process. It should permeate all of the different activities and link them together under a common goal, producing quality software. Figure 2.4 shows the relationship of quality assurance and the various techniques, activities, and procedures, as well as, the artifacts produced as a result of these activities of software manufacturing.

The quality system should contain standards that have been developed both within the software development organization and those that have been accepted by the software development community. These standards are discussed in Section 2.5.2. In addition to standards, one should also utilize a process that reinforces the concepts of quality and provides a framework for integrating standards and procedures into the software development organization.



[Frank, et al., 1996]

Figure 2.4 Quality Assurance's Central Role

As one can see from Figure 2.4, quality assurance impacts on each of the elements represented by the various circles. Project management aims at producing quality software by a process that focuses on quality. Changes are controlled so that quality is maintained in the software. Even configuration management is a control process that helps assure that the correct software is included in the final product. Documentation is crucial to software, particularly the documentation

related to specification and requirements. The end-user documentation, again an important contributor to software quality, must be both understandable and complete. As one can see from this brief discussion, each of the elements in the figure is connected by quality assurance. The quality assurance function oversees the elements outlined in this figure, as well as, manages the testing of the software during and after development.

Software quality assurance correlates highly with traditional product quality assurance in that one is working with a process that produces an end product. For this reason, one can rely on the existing literature and practices that have been developed for the manufacturing industry. [Card, 1990] expands the traditional concept of software quality assurance to 'software quality engineering (SQE).' 'SQE extends the product concerns of traditional software quality assurance to encompass process and engineering issues.' For Card SQE includes the following functions:

- defining and customizing the baseline software process,
- inspecting software products,
- auditing the software process,
- monitoring process quality,
- monitoring product quality,
- designing producible products, and
- improving the software process over time.

Card sees SQE as an integrated approach to software development that does not focus on a single quality assurance function but instead includes the entire development process. This concept is in synchronization with the ideas presented in this thesis.

2.5.2 Standards for Software Development and Quality Assurance

Standards should play an important role in the development of quality software. Software standards provide a framework under which software should be developed and tested. There are

industry-wide standards and company-based standards, as well as, defacto and de jure standards. Organizational standards, in order to be effective, must be both carefully developed and documented and then enforced. A standard that is not enforced is equivalent to not having a standard.

In regard to industry-wide standards, there are a number of standards in existence today that relate to software engineering and software testing. Table 2.4 provides an overview of those standards. Included in this list are standards that apply to quality assurance in general. Presently, there are 27 standards approved by the IEEE that relate to software development.

Table 2.4 Quality and Software Standards

	ANSI/IEEE Standards
ANSI/IEEE Standard 730	Standard for Software QA Plans
ANSI/IEEE Standard 828	Standard for Software Configuration
ANSI/IEEE Standard 830	Guide to Software Requirements Specifications
ANSI/IEEE Standard 1028	Standard for Software Reviews and Audits
ANSI/IEEE Standard 1012	Standard for Software Verification and Validation Plans
ANSI/IEEE Standard 1074	Standard for Developing Software Life Cycle Processes
	ISO Quality Standards
ISO 9000	Quality Management and Quality Assurance, Guidelines for Selection and Use
ISO 9001	Quality Systems - Model for Quality Assurance in Design, Development, Production, Installation, and Servicing
ISO 9002	Quality Systems - Model for Quality Assurance in Production, Installation, and Services
ISO 9003	Quality Systems - Model for Quality Assurance in Final Inspection and Test
ISO 9004	Quality Management and Quality System Elements Guidelines
ISO 9000-3	Guidelines for Applying Quality System Requirements of

In addition to the existence of the above standards for the management of quality assurance and testing, there have been a number of initiatives directed toward improving software quality and the software development process. Those efforts are aimed at helping software development organizations evaluate their current processes and identify those areas where improvements need to be made. Additional standards are included in the references to this thesis.

2.6 Frameworks for Software Process Improvement

As the software development process is evaluated and the software under that process is tested for functionality and measured against established measures of quality, changes in the development process may be warranted. To evaluate the process, as well as determine what

elements should be contained in a quality process, one must have access to accepted process improvement guidelines. The Capability Maturity Model, Trillium, Belcore TR-179, Bootstrap, and ISO SPICE are all examples of how one can evaluate and establish a software process that contributes to the production of quality software. These models and guidelines are discussed with comparison made between them.

2.6.1 Capability Maturity Model (CMM)

The Capability Maturity Model, developed by the Software Engineering Institute (SEI) at Carnegie Mellon University, comprises a means of evaluating a software development organization's software process. The CMM has five levels of maturity by which the process is judged. Each level is composed of key process areas and key practices. The level of maturity ranges from Level 1 - Initial Development Process to Level 5 - Optimizing, where there is an organization-wide focus on continuous improvement. Table 2.5 lists the key process areas for each of the five levels of the CMM.

Software quality assurance is explicitly addressed as a key process area at the Repeatable Level of Maturity (Level 2). Software quality management is the concern of the key process area at the Managed Level of Maturity (Level 4). Software testing will be included at the Defined Level (Level 3) since it is part of a standardized software development process. 'A defined software process contains a coherent, integrated set of well-defined software engineering and management processes.' [Paulk, et al., 1993]. It is logical to think that proper software testing is just another component of a well defined software development process.

Table 2.5 Capability Maturity Model

LEVEL	KEY PROCESS AREA
1. Initial	
2. Repeatable	Software Configuration Management Software Quality Assurance Software Subcontract Management Software Project Tracking and Oversight Software Project Planning Requirements Management
3. Defined	Peer Reviews Intergroup Coordination Software Project Engineering Integrated Software Management Training Program Organization Process Definition Organization Process Focus
4. Managed	Software Quality Management Quantitative Process Management
5. Optimizing	Process Change Management Technology Change Management

[Humphrey, 1989]

[Humphrey, 1989] devotes a chapter specifically to software testing. Humphrey limits testing to the actual execution of software to find faults. In order to effectively implement a testing function within a software development process one must plan for testing, develop the test cases, report the results of each test execution, and thoroughly analyze the results. For testing to be adequate, it is necessary that the testing function be organized and managed. Leaving testing to programmers results in less-than-complete testing.

2.6.2 Trillium

The Trillium Model [Coallier, 1995], developed in a cooperative effort of Bell Canada, Northern Telecom, and Bell-Northern Research, focuses on the means necessary to develop and manage a continuous improvement program. Trillium contains essential practices that can be used to improve an existing software development process. Trillium has 8 Capability Areas, (Organizational Process Quality, Human Resource Development and Managements, Process, Management, Quality System, Development Practices, Development Environment, and Customer Support) with each having multiple levels of maturity. By using Trillium's evaluation to create a profile of existing capabilities, one is able to identify areas within the organization that could benefit from improvement.

2.6.2 Belcore TR-179

The Bellcore TR-179 document [Bellcore, 1993], developed by Bell Communications Research, is comprised of a set of 25 generic requirements that focus on both the need for supplier accountability and improved quality in software used for telecommunications networks. TR-179 applies to both the process and the actual software developed by that process. The requirements of the Bellcore Model were designed to correspond with the ISO 9000-3 guideline structure. Table 2.6 provides a list of the 25 requirements along with their major categories.

Table 2.6 Quality System - Framework

TR-179 REQUIREMENT
1. Management Responsibility
2. Quality System
3. Internal Quality System Audits
4. Corrective Action
5. Quality Improvement
QUALITY SYSTEM - LIFE CYCLE ACTIVITIES
6. General
7. Contract Review
8. Purchaser's Requirements Specification
9. Development Planning and Project Management
10. Quality Planning
11. Design and Implementation
12. Testing and Validation
13. Acceptance
14. Replication, Delivery, and Installation
15. Maintenance
QUALITY SYSTEM - SUPPORT ACTIVITIES
16. Configuration Management
17. Document Control
18. Quality Records
19. Measurements
20. Rules, Practices, Conventions, and Standards
21. Tools and Techniques
22. Purchasing
23. Included Software Products
24. Training and Human Resources
25. Purchaser Operational Assistance

[Bellcore, 1993]

Testing is included in the Life Cycle Activities, number 12 on the list in Table 2.6. Note, the list is process independent; thereby, the order of the elements has no correlation to where each belongs in a development process. However, it is noted that the list does give one a sense of feeling for the Waterfall Method.

2.6.4 Bootstrap

The Bootstrap method [Kuvaja, et al., 1994] was developed by the European Strategic Program for Research in Information Technology (ESPRIT). The purpose for developing Bootstrap was to create an assessment tool for evaluating the software development process, along with a plan for improving that process. Bootstrap focuses on three primary areas 1) organization, 2) methodology, and 3) technology. Bootstrap is similar to SEI's CMM; however, instead of using only five levels of maturity, Bootstrap proposes a separate level of maturity for each attribute of the process model.

The issues of testing and software verification and validation are classified in two different areas of the methodology. Under the life cycle dependent component of Bootstrap the issues relating to software testing are addressed. Here, Bootstrap is concerned with software implementation and testing, software integration and testing, and system integration and testing. Under the Life Cycle Independent portion of Bootstrap, the general issues of Quality Assurance, Verification, and Validation are addressed. Within the three major areas of Bootstrap, testing falls under the Organization and Methodology areas.

In regard to each of these issues, Bootstrap focuses on the evaluation of the process and from this a plan for improving the software development process is constructed. As with the other life cycle improvement plans, Bootstrap is process independent, not endorsing a particular life cycle model.

2.6.5 ISO SPICE

The International Standards Organization (ISO) has developed a set of standards for use in evaluating the software development process under the SPICE (Software Process Improvement and Capability dEtermination) project. SPICE includes components taken from other process improvement approaches, including CMM, Bootstrap, and Trillium [Konrad and Paulk, 1995].

SPICE attempts to develop a means of measuring process maturity without prescribing a particular approach to process improvement.

SPICE extends its focus to include people, technology, management practices, customer support and quality, as well as software development and maintenance practices. The SPICE standards are designed to be used for process assessment, process improvement, capability determination, and qualification and training of assessors. SPICE examines a number of different processes which are divided into a set of guides: Baseline Practices Guide (BPG), Process Assessment Guide, Process Improvement Guide, and Process Capability Determination Guide. Each of these is helpful in determining the current state of an organization against the Baseline Practices which examine the customer-supplier relationship, the engineering process, the project process, the support process, and the organizational process. Within each of the major baseline practices there are more detailed subcategories used to measure maturity of the software development process [Emam, et al., 1997].

SPICE is not prescriptive in the area of software testing, however, there are specific areas within the BPG which address the issue of testing. Testing is measured under the Engineering category and under the Support Category where quality assurance is examined.

2.7 Software Testing

Software testing is a crucial part of the software development life cycle. With testing one is attempting to identify the existence of defects. Once identified the software must be modified and retested. It is a destructive process that focuses on fault finding. Software testing cannot show that the software is without defect, only that defects exist. However, it can help develop statistics that show that the frequency of fault identification has declined. These statistics can then be used as an indicator of fewer remaining defects in the software. There are certain types of tests which can show the absence of certain, but not all, types of faults. In addition, testing can be used as

a measure of the quality of a particular piece of software. One must be cognizant that testing is directed at individual programs, sub-programs, and entire software systems.

The definition of software can be expanded to include analysis documents, design documents, requirements, and test cases. With this expanded view of software one can begin to look at software testing in a new light which includes testing of these other elements of the design process. This expanded view of testing brings attention to the need of testing the requirements for incompleteness and ambiguity.

In testing the documentation components it is apparent that one must first create the documentation. Regardless of the development life cycle being used there must be a strict adherence to the method else there will be areas where the required tasks are incomplete.

Even an expanded definition of software testing which includes the testing of the analysis, design, and test documents, does not adequately cover the entire life cycle. One must also be willing to evaluate the actual process by which the software is being developed. Compliance audits can be included that are used to identify areas where the implemented software development life cycle deviates from the stated process.

Finally, one must evaluate the actual process to see if it is meeting the needs of its customers. With software development, as with other product development processes, there are both internal and external customers. Both subjective and objective measurements must be taken that can be used to identify potential flaws in the process. Just complying with a set of standards and requirements does not in any way assure the effectiveness of the process.

2.7.1 Role of Software Testing

The testing life cycle corresponds to the development life cycle. In fact, testing actually consists of two interrelated cycles. Those cycles are 1) the decision-making process for determining which

tests to perform and the development of the tests and data sets and 2) the carrying out of the tests and the evaluation of the results. Execution of the tests is closely aligned with the overall development life cycle. Testing is performed throughout the individual phases of the object-oriented life cycle. Software testing can be viewed in the same light as software development. [Humphrey, 1989] identified three levels for process models. Those levels are 1) The Universal which represents a high level overview of the process, 2) The Worldly which corresponds to the working level, and 3) The Atomic which provides additional detail to the Worldly View. Table 2.7 provides the activities that occur at each of the equivalent levels of testing during software development.

Table 2.7 Levels of Abstraction for Software Testing

Level	Activities
Universal	Testing defined and goals established, managerial planning for software testing completed at this level. At this level there is an overall view of where testing will occur.
Worldly	Attention is drawn to the implementation cycle. Specific testing techniques, test schedules, and resources, system and acceptance tests are defined and plans for each are developed. Data is collected at this level for the initial core test cases.
Atomic	Testing occurs at this level. Creation of test suites, execution of the software, and evaluation of the results is carried out here.

Specific tests, as well as reviews, are used to measure both the behavior and the functionality of the software under development. Each phase of the life cycle must have a corresponding testing phase. The testing phases are all linked together, as well as, integrated with the actual software development life cycle. Feedback from each of the tests has a direct impact on the next phase of software development. Failures resulting from test execution require that one repeat that particular phase of software development.

The use of testing is analogous with the continuous data collection and analysis associated with

the flight of a rocket. Suppose one wants to send a rocket to a distant planet. If one merely builds the rocket and then fires it toward the planet it may or may not hit the desired target. If, on the other hand, careful testing occurs, starting at the analysis and design phases, then the rocket is more likely to be at least fired in the right direction. Failure to test the design, however, may result in either too powerful a rocket, too small a rocket, or a rocket with some other critical limitation.

Once the rocket is fired one must continually collect and analyze flight data. The location and trajectory data must be compared with the known route to the target planet. A slight deviation from the planned route must be quickly corrected if one is to achieve the ultimate goal of landing the rocket on the planet.

A variance of only a fraction of a degree from the desired route is relatively easy to correct if that deviation is detected early in the flight; however, waiting until later in the trip could conceivably place the mission in jeopardy since it may be virtually impossible to make a correction of such magnitude. The following drawing illustrates the need for early detection and correction of any deviation from the specified route, regardless of the degree of variance.

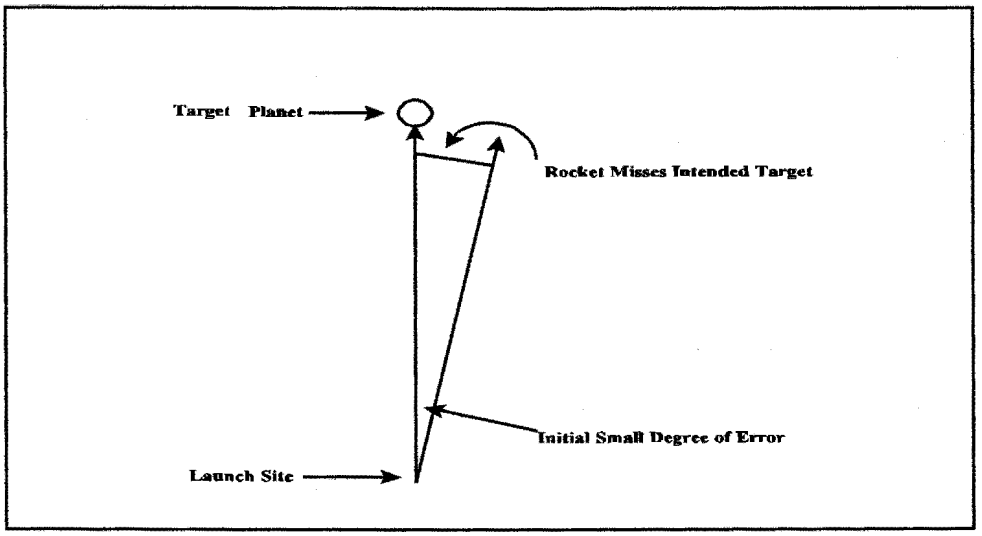


Figure 2.5 Planetary Exploration

Software development is much like this illustration. Testing often throughout the life of the software development will result in a product of higher quality developed at a much lower cost. Waiting until the software is finished to begin testing is like waiting until the rocket is supposed to land on the planet and then looking to see where it actually is. Both situations are doomed for failure.

The issue is not to attempt to find a solution after a major problem has been discovered but to avoid the problem in the first place. Minor adjustments are easier to make and less costly. Firing the rocket in the wrong direction may be such a major fault that no correction will result in the success of the journey. Designing a software solution to the wrong problem will be just as great a disaster.

2.7.2 Quality Assurance Through Testing

The QA Group has a vital role to play in determining what quality actually means in terms of software. It is the responsibility of the QA function to see that all quality issues are resolved

during the various phases of the development life cycle. Testing, on the other hand, is more related to verifying that indeed the product was tested and that it meets its requirements, both functionally and qualitatively. A software product that is thoroughly tested and produces correct results may in fact be of poor quality, either aesthetically and/or operationally. Testing is not a substitution for quality assurance. In fact, one can go as far as to say that the testing component is merely a subset of Quality Assurance.

Testing aids in the development of quality software by assisting in the identification and removal of defects, while at the same time, providing a means by which the end user can be effectively represented during the development process. As software is developed, the testing group evaluates the software against the standards and requirements contributing to the system design and implementation. Software is tested at the unit, subsystem, and system levels. Expanding testing to include code walkthroughs and inspections further contributes to the quality of the final product in that additional defects may be identified and removed before final acceptance tests are performed.

In order to fully comprehend the various elements of testing as applied to the life cycle, the work of [Thornton and du Plessis, 1995] is used as a foundation. In their work they define the elements of quality assurance as they relate to the revised spiral model for software development.

[Hetzel, 1988] focuses on the need to modify the life cycle by integrating testing into each phase. His viewpoint, endorsed by the author of this thesis, identifies testing as a key component that permeates the entire life cycle. Table 2.8 shows the integration of testing into the traditional waterfall life cycle.

Table 2.8 Integration of Testing Into the Development Life Cycle

Project Initiation
Develop Broad Test Strategy Establish Overall Test Approach and Effort
Requirements
Establish Testing Requirements Assign Testing Requirements Design Preliminary Test Procedures and Requirements-Based Tests Test and Validate Requirements
Design
Prepare Preliminary System Test Plan and Design Specifications Complete Acceptance Test Plan and Design Specifications Complete Design-Based Tests Test and Validate the Design
Development
Complete System Test Plan Finalize Test Procedures and Any Code-Based Tests Complete Module or Unit Test Designs Test the Programs Integrate and Test Subsystems Conduct the System Test
Implementation
Conduct the Acceptance Tests Test Changes and Fixes Evaluate Testing Effectiveness

[Hetzel, 1988]

Testing must be performed throughout the entire life cycle, regardless of the chosen development model. Many software development groups attempt to perform testing at a later stage in development, well after the design and specifications have been established. This delay has the potential of adding to the cost and time necessary for developing the software. The fundamental

purpose of testing is to find defects, not to prove that the software works correctly. In fact, it is virtually impossible to prove the correctness of software. Locating faults, on the other hand, is possible.

During testing, data are collected and analyzed thus allowing the testing group to provide feedback to the development team. These data are used to determine if changes are needed in the software, the design, or both. Feedback, also provided to the testing team, is used to make modifications to the test suites and to the types of tests being run.

2.8 Summary and Conclusions

The quality of software is composed of a combination of both the standards and the method by which the software is designed and developed. One must first begin with a workable definition of software quality. From here one adds both internal and external standards along with the user's requirements. By utilizing a formal development life cycle process, along with quality assurance which includes software testing, one is able to develop quality software. Continuous process improvement must be coupled with quality assurance that examines both the product and the process. Various software process models were reviewed in this chapter. Of particular interest is the iterative approach to object-oriented software development. Combining the iterative approach and the spiral model of Boehm led to interest in the Revised Spiral Model for Object-Oriented Software Development of van der Walt and du Plessis. This model provides a solid theoretical and practical foundation for understanding the software development life cycle as it is applied to object-oriented software. The Revised Spiral Model for Object-Oriented Software Development combines an acknowledged software development life cycle with sound software engineering concepts underlying object-oriented development and holds promise in building quality software.

In regard to the modeling language, UML was chosen for use in this thesis due to its wide

acceptance for visual modeling of object-oriented software. UML, due to being process free and containing all of the meta-primitives necessary for modeling object-oriented software, merges with the Revised Spiral Model to provide a complete process for developing object-oriented software.

CHAPTER 3

Integration Testing of Procedurally-Based Software

CONTENTS

3.1 Introduction

For one to understand the issues surrounding the testing of object-oriented software it is imperative that one have a working background in software testing in general. Concepts have been developed over the years by individuals interested in the testing of software and software quality assurance. Since the primary development model for software during this formative phase of testing philosophy and techniques was the Waterfall method, coupled with procedurally-based software development, it is necessary that one look at traditional software testing and determine which, if any, of its concepts and techniques are applicable to object-oriented software.

This chapter examines the various techniques used for identifying faults and inconsistencies between modules of a program or system. An evaluation is provided which aims at assisting the reader in understanding the purpose of integration testing and the various techniques available for performing integration testing.

Integration testing accounts for approximately 9% of the defects found in software [Beizer, 1990]. Other researchers have reported a varying range of percentages in their classification of defects [McConnell, 1993]. [Harrold and Soffa, 1991] reported that some studies have raised the percentage for integration type faults to as high as 40%. Regardless of the actual contribution of integration defects to the overall number of faults, this type of defect tends to be critical because these faults are often discovered late in the development cycle and have a high correction cost. For this reason it is important that one understands the nature of faults found during integration and system testing. Furthermore, in order to discover this type of fault it is equally important that tools and techniques be available to perform integration testing.

3.2 Software Testing Phases

Traditional software development consists of the following phases: 1) problem definition, 2) analysis, 3) design, 4) implementation/coding, 5) testing, 6) installation, and 7) maintenance. Whether one uses the Waterfall Model, Boehm's Spiral Model, or some other model of the development life cycle, there are specific types of testing that must take place. These tests can be classified as 1) unit testing, 2) integration testing, 3) system testing, and 4) acceptance testing. Testing can be classified into two major groups, non-code based testing and code based testing. Non-code based testing focuses on the testing of the analysis and design components of the system. Evaluation and checking for inconsistencies and ambiguities within the system requirements is an example of non-code based testing. Code based testing, on the other hand, requires the examination and/or execution of the actual software that implements the software

design. The various types of tests are reviewed next.

3.2.1 Unit Testing

Unit testing focuses on the internal operation of the code within a unit. Under procedurally-based software the unit can be identified as a procedure, a paragraph, a function, or a subroutine. Because unit testing entails examining the execution of the statements within the unit, this form of testing is often called White Box Testing or structural testing. Unit testing is limited to internal executional flow through the unit, examining how different threads are followed and if correct decisions are made as the code executes. Traditional unit testing is the responsibility of the individual who coded that unit. Units are tested before they are released to the build.

3.2.2 Integration Testing

Integration testing, as defined by [Beizer, 1984], focuses on 'showing inter-element consistency under the assumption that the elements themselves satisfy element requirements and have passed element level testing.' Element level testing corresponds to unit testing. Units can be identified as defined above or can be viewed as grouping of similar components that form sub-systems of the program. Integration testing is concerned with the functionality of the software as exemplified by the interaction among the units that makeup the software, e.g., calling the units in the proper sequence with the right arguments and receiving the correct responses.

Integration testing, in fundamental terms, relates to the testing of individual units as they work together to perform tasks within the system. Integration testing focuses on the external interfaces of these units and the arrangement in which units collaborate. For integration testing to be carried out one must examine the interfaces, the messages passed to units when they are called, and the messages returned after a unit has completed its function.

3.2.3 System Testing

Having completed unit and integration testing subsystems are combined into larger components. Eventually the entire system is submitted to testing. System testing is concerned with the overall operation of the software along with examination of external interfaces. System testing uses the entire software product and is aimed at revealing defects that cannot be produced when executing sub-components. System testing focuses on such issues as security, performance, and usability.

System testing emphasizes the external interfaces that the individual sees when using the software. Instead of being concerned about the construction and internal makeup of the software, system testing measures the actual software against the design. The overall functionality of the software and its response to input criteria are tested against system specifications. Test cases are designed to test the accuracy of the software under varying conditions, the ease of use of the software, and the performance of the software as related to response times. Many of the aspects of software quality are examined when one is involved with system testing. Acceptance testing, more than examining the accuracy of the software, is concerned with such issues as ease of use, flexibility, and other quality elements as discussed in Section 2.2.

3.3. Goals of Integration Testing

Since this research is concerned with integration testing, it is necessary that the discussion of integration and integration testing be expanded. To understand integration testing, it is important to know what types of defects one is looking to uncover and what types of tests and techniques are most effective in that endeavor. The following section analyzes both faults and tests.

As stated above, integration testing is concerned with the sequence of unit interactions along with the passing of arguments and the returning of values. One must test the operation of the software

against known requirements and specifications in order to measure the software's operation. Integration testing is an iterative process which must be continued until the entire program or system has been thoroughly integrated and tested. [Beizer, 1984] provides an outline of the target of integration testing. These elements, listed in Table 3.1, are the key issues that should be examined.

Table 3.1 Issues Related to Integration Testing

Range	here the extremes of values, along with excluded values, are tested
Type Compatibility	a strongly typed language should catch many of these inconsistencies; however, one should test to catch any incompatible data types
Number of Parameters	if the number of variables is fixed, then one should test to make certain that the required number is being provided; if the number of arguments is variable, then specific tests will have to be developed for testing of different numbers of values under specific conditions
Input/Output Parameters	the means by which values are passed should be tested; there may be instances where the variables are read only, however, they may be passed by address and the passed object modified
Object Order	the order of the variables must be known to the receiving subroutine; if a different order is accepted, then there must be a means by which the receiving parameters can be alerted of the specific order
Method of Transfer	parameters can be passed by a variety of means, both direct and indirect; compatibility of the manner in which a parameter is passed must be tested with each subroutine call; multiple layers of indirection can lead to confusion and incorrect results
Variable Element Name in Call	the actual subroutine called can be dependent on a control

[Beizer, 1984]

3.4 Integration Testing - Tools and Techniques

For the purpose of developing a thorough understanding of integration and system level testing, the following sections focus on techniques and tools required to perform that level of testing. Thereafter, system level testing is addressed along with how it relates to integration testing.

[Beizer, 1984] provides the basic foundation for integration testing of programs developed in procedural languages. His book still remains as a major reference in the field of software testing. He states that in order to perform integration testing one must first have an acceptable understanding of the functionality of the software to be tested. This understanding requires that there be adequate documentation regarding both the user requirements and the structural design of the software.

Once the necessary system specifications are acquired one is then ready to begin to develop a set of tools and to derive the data from the software in order that it can be compared with the system requirements. Basically, there are two important questions to answer: 1) Does the program allow the subroutines to be called without defect and do these subroutines return the required data types?, and 2) Does the software produce the desired results? Each of these two issues is extremely important. As mentioned earlier, a program may be compiled and linked without flaw; however, the end results may not be what was originally agreed upon in the specifications. Without detailed specifications, integration and system testing will be less-than-complete.

In regard to the internal structure of the software, it is imperative that there be a clear understanding of how the program is constructed in order to compare the program structure with the software design specifications. Oftentimes the user requirements or the technical specifications are incomplete making it necessary that there be a means by which the individual performing integration testing can obtain an understanding of the software at hand. One of the fundamental ways of developing this information is by creating a Call Graph of the program. A Call Graph presents, in either a graphical or textual form, all of the calls to subroutines and lower-level routines. Such a Call Graph can then be used to compare the actual implementation with

the original design to determine if the proper subroutines were called, and in the correct order. If the software requirements are not available the Call Graph is helpful in developing necessary system documentation.

Call Graphs can be produced in a number of ways. The most elementary means is by manual inspection of the source code. This process is time consuming and in many cases may actually miss some of the routines. However, by performing such a task, one is able to examine the code directly, much like a code review, and may discover weaknesses in the program. This form of Call Graph creation is one type of a Static Call Graph, since the program is not executing while the data are collected.

A second type of Static Call graph can be produced by using the program being analyzed as data for input into a second program capable of understanding the specific language in which the source program is written. The second program then produces a Call Graph of the various subroutines as they are called. This automated generation of the Call Graph can be produced either from the original source code or from the assembly code produced by the compiler. [Bergman-Terrell, 1991] provides a way in which a Call Tree can be created from the assembly language output of a program compiler. His program is designed to provide a textual representation of a Call Tree. Additional Call Tree generators and program profilers have been developed by academic researchers and software development practitioners. Several of these programs are commercially available such as 'The Documentor' by WallSoft Systems, Inc., 'Source Print' by Powerlint Software, 'C-DOC' and its related products by Software Blacksmiths. Two additional profilers have been presented in Doctor Dobb's Journal. These two profilers, not entirely related to integration testing, however, do provide important information about the structure of a C program and are quite helpful in the actual development of a Static Call Tree [Hymowech, 1988][Nutter 1988].

A third method of generating Call Trees is based on the actual execution of the software and the

collection of data as the program runs. This type of Call Tree development, referred to as dynamic analysis, provides one with snapshots of the program as it executes.

The most common manner of implementing a Dynamic Call Generator is through source code instrumentation. Here the data producing code is added to the program being tested. Instrumentation requires that the data generating statements be inserted either manually or by having a second program read the source code and modify it.

The simplest way is to edit the program under consideration and insert simple statements, like *printf* in the C language, within each subroutine. As the subroutine is called, the *printf* statements output the information concerning the function that is being called, along with the values of the parameters. Additional *printf* statements may be used prior to any return statement to indicate the value(s) being returned. After the source code is instrumented the program is compiled, linked, and executed. As the program runs data is output from each subroutine as it is called. The output of the instrumented program is then analyzed by another program which actually creates the Call Tree. The combination of data generating and data processing programs can be modified to produce either textual or graphical output, or a combination of both, for analysis and documentation purposes.

To aid in data collection, macros can be used to instrument the program. Whenever the compiler is called with the necessary macro definition option, the program is compiled with the instrumentation statements. This procedure allows the programmer to leave the statements in the program and simply turn on the data producing code whenever desired.

Instrumentation has the advantage of being able to produce data as the program executes. There are problems associated with this technique, however. For one, the program is actually modified by the instrumentation. This modification results in additional statements being executed resulting in timing delays. This change in the timing of the program prohibits a true evaluation of the

performance of the software. After the instrumentation code is removed, the operation of the program will actually change with the potential of allowing additional defects into the software. For this reason, the use of data production statements which remain in the final source code is recommended.

A symbolic debugger may also be used to trace through the execution of a program. Data collected during the tracing of the program can, with some degree of effort, produce a representation of the call structure of a program. The modifications to the program may be non-existent, in this instance, or may be produced by the compiler when the program is compiled in debug mode.

In addition to the problems related to the modification of the program under test, there are limitations associated with the Dynamic Call Tree being generated by executing the program. Data from the entire program may not be displayed since it is virtually impossible to test all of the potential paths through a program. For this reason production of both a Static and a Dynamic Call Graph is important for the understanding of the actual operation and construction of a program.

In conjunction with the development of the Call Graph, there are a number of other helpful representations of a program. [Perry, 1991] discusses different static and dynamic analysis techniques. He focuses on the use of flow analysis for the analysis of both data and program control. Data Flow Analysis is used to identify undefined or nonreferenced data elements. Control Flow Analysis is used to analyze the behavior of the program. Both of these types of analysis are derived from different versions of a graph developed by statically examining the program. It must be understood that both types of flow analysis require more than a simple graph of the program that shows which subroutines are called and in what order. In Data Flow Analysis the tester is concerned with 'tracing the behavior of program variables as they are initialized and modified while the program executes' [Perry, 1991]. In developing such information one has to

know when a specific variable is defined, initialized, and used in either the right side or left side of an expression.

[Beizer, 1984] identifies other types of techniques and attendant representation schemes for static analysis. These techniques are the Data Dependency Graph, the Data Dictionary, and the Process Dependency Graph. The Data Dictionary is crucial to the identification of both global and local data elements. He states that it is very difficult to perform integration testing without this dictionary. Various compilers are capable of generating the Data Dictionary which can provide such detail information as the name of the data element, the line on which it is defined and the type and size of the variable. In addition to this information, many compilers provide a list of all of the lines in which a specific variable is referenced. When utilizing a CASE tool for software development, the Data Dictionary is included as a critical component of the development products.

All local variables should have been tested during unit testing. During integration testing, data that is global to several subroutines is tested. It is important to have a list of both the local variables and the global variables within a program. Depending upon the scope rules for a particular language under consideration, the declaration of a local variable with the same name and type as a global variable may result in an unintentional fault.

The Data Dependency Graph, seen as an extension to the Call Graph, provides the user with information about the data items involved in particular calls. Here one is able to determine whether the call modifies a global variable or not. In this case the arguments and related formal parameters of the subroutine must be examined to determine whether a read or write call was made. A read call does not effect the global variable, where a write call does. However, there are instances where a read may actually modify a global variable. A write call, though, must be thoroughly examined to make certain that there is not some inadvertent side effect that impacts on more than the desired variable or variables.

The Process Dependency Graph [Beizer, 1984] considers how elements are transformed by the individual processes. This type of analysis is severely limited since it is very difficult to determine various data relationships when only using static analysis. However, the Process Dependency Graph can be helpful when considering dynamic analysis techniques.

Dynamic analysis of a program can greatly improve the information about how a program works. Following this method a more detailed understanding of the nature and structure of the program in question may be developed. A common means of performing dynamic analysis is to first examine the program with a static analysis tool. Then the individual subroutines are seeded with data collection statements, the program is executed, the data collected, and a second program is run, using the output from the instrumented program as input, to produce the various graphs and analyses.

Having considered the necessary analysis tools, static and dynamic call graphs, data dictionaries, and process graphs, along with a set of elements that should be tested, the issue of knowing when a given program has been tested warrants investigation. [Beizer, 1984] states that integration testing cannot be considered complete until every subroutine has been tested. For him, static analysis is not sufficient for the development of a set of tests; dynamic analysis provides a better means for the development of these tests since:

> 'an element (subroutine) cannot be considered integrated until every path in its real, dynamic, call graph has been exploded under the test' [Beizer, 1984].

There are two parts of integration testing that have to be completed: 1) every subroutine in the Dynamic Call Tree must be called on some path in the Call Graph, and 2) every called subroutine must be called by all possible callers.

[Beizer, 1984] recognizes the need for extensive integration testing. In fact, he states that:

> 'a program cannot be considered successfully integrated until all links in the Call Graphs that lead to them have been tested. If proper path testing is done for all elements at all levels, and path coverage is provided, then when all calling elements have been tested, all calls to a routine such as B will have also been tested and Call Graph cover will have been achieved.'

From this statement one can see that integration of a program or system can be extremely complex, particularly if one has a set of routines which can be called from numerous other routines in a variety of different ways.

Because of the potential for excessive complexity within a program, multi-entry and multi-exit routines add to the need for additional testing that entails testing all entry points and exit points. If there are multiple exit points within a routine one should treat each exit point as a separate routine and should test it accordingly. This complexity increases the number of tests which have to be run.

Additional problems can exist which are not readily found during unit testing. Specifically, there can be either data corruptions or data residues which are a result of improper coding techniques. These defects are more likely to be discovered by the use of code reviews and walkthroughs. Walkthroughs and reviews allow the programmers and quality assurance personnel to work together to examine the software products, including both the code and the related documentation, and to locate areas where there are defects and weaknesses. Examining the software in such a detailed manner often reveals faults which are overlooked during routine software testing, particularly in regard to quality issues.

3.4.1 Integration Strategies

The actual integration of programs is directly linked to the integration testing of software.

Integration testing strategies today still remain similar to those developed during the 1970s and 1980s. Essentially these strategies can be classified as top-down, bottom-up, big-bang, and a combination of these strategies. Integration testing should begin when one combines two units and continues until the entire system has been integrated.

3.4.1.1 Top-Down Integration/Testing

With this technique, integration starts with the topmost module. All other modules are initially implemented as stubs. After the top module is thoroughly tested, each stub is replaced one at a time with the actual code and the testing process is repeated. This incremental process is continued until all stubs have been replaced and the entire program or system has been tested. Under the "pure" model of top-down testing, no unit testing is carried out. [Beizer, 1984] identifies a number of myths surrounding this form of integration and testing.

Recognizing these myths is helpful in understanding that integration testing does not have a single technique that is applicable in all instances. One has to use different views of the software as it is integrated in order to develop a comprehensive understanding of how the different software components work together. This understanding of the software allows one to develop both test cases and the methods used in integration testing.

Table 3.2 Integration Myths

Top-down design limits the complexity at any one level, and therefore, top-down integration and testing limits testing complexity.
Complexity decreases uniformly from top down.
There is a single calling program.
Testing with stubs is easier than with real routines.
The system is the best test driver.
Top-down testing is a natural adjunct to top-down design.

[Beizer, 1984]

Each one of these misconceptions can lead the tester to not find all of the defects within a program. This emphasizes the need for additional techniques for integration testing. However, this technique does have some reasonable uses [Beizer, 1984].

Table 3.3 Top-down Integration Utilization

Mechanical integration - compilation, linking, loading.
Stubbing of units to reduce testing complexity.
Top-down testing of control structures.
Sub-element interface testing.
Element-level structural and functional testing..

[Beizer, 1984]

Top-down integration testing is one technique that is commonly taught to beginning programming students. Instead of attempting to write the entire program/system at one sitting, the student is taught to develop the driver module for the program and then to stub each of the additional sub-modules of the program. As each module is compiled, then the stub is replaced and that unit is integrated and tested. Top-down testing can assist one in finding problems with interfaces and control. These issues are certainly of concern to the tester.

3.4.1.2 Bottom-Up Integration/Testing

The next approach is bottom-up integration and testing. Here one starts at the bottom most level of the Call Tree. The individual units are integrated to form larger units. This process continues until all units have been integrated and tested and the entire program or system has been tested. In order to perform bottom-up testing it is necessary that one develop driver routines that are used to call the subroutine and pass arguments to the routine.

[Beizer, 1984] points out several misconceptions inherent in bottom-up testing:

1) if the units are thoroughly tested and carefully integrated then the whole program/system need not be tested

2) complexity increases from the bottom to the top

3) once a bug is corrected at the lower level, it remains corrected throughout the entire program/system

4) test drivers are easy to build.

One of the major concerns with test drivers is that one must be careful in constructing them. These driver routines must be thoroughly tested. A fault in the driver routine can result in a new defect being introduced in the software that the driver routine has been built to test. A problem of bottom-up testing is that the testing is being performed in a simulated environment instead of

using the actual driver routines of the software implementation. Problems not identified during the testing phase may appear after the system is actually implemented.

3.4.1.3 Big-Bang Integration/Testing

Big-bang testing, like its name implies, is a discrete event. One waits until the entire program or system is completed, then the program is run and one tests to see if any problems occur. As most programmers know, nothing works properly the first time, if it works at all.

The reservations about using big-bang testing relate to the fact that it is widely considered by many persons that big-bang testing is really testing without any plan or strategy. However, in many cases this is exactly how software is actually tested. As soon as enough code is written to have a program that is executable, the program is run and the results are used for correcting the problems which were encountered.

This method is far less-than-satisfactory. If there are faults found then there is some validity to the method. With big-bang integration testing one can save a lot of time because the individual units are not integrated and tested one at a time; however, the time spent in locating defects and removing them far outweighs this trivial time savings. One should recognize the advantage of first testing units, integrating them, and then testing by using top-down, bottom-up integration testing.

3.4.1.4 Unification of Top-Down, Bottom-Up, and Big-Bang

[Beizer, 1984] concluded that the most appropriate method of performing integration testing is achieved by having a combination of these techniques. His approach to integration and testing is as follows: 'Bottom-up the small; top-down the controls; big-bang the backbone and refine.' His seven step approach is designed to find as many of the problems associated with integration of modules into a workable system.

In bottom-up the small, he feels that the small assemblies of units should be tested before the control structure becomes too complicated. This step can identify problems which are related to the actual modules before one has such a difficult time determining the cause of program failure.

As the control structures become more complicated, he feels that top-down testing is best suited. One should not attempt to top-down test too many levels, preferably only two. The other units at the lower level should be stubbed. Once the top-down testing is completed then the stubs can be replaced with the actual code.

The backbone of the program or system should be tested in a slightly different manner. In this case, big-bang testing is suitable. The back-bone must first be defined, that is, that part of the system that must be available for testing of the subunits: the input, output, and memory management portion of the system. After thoroughly testing all of the units of which the backbone is composed, then compiling, linking, and executing the backbone is a good way to discover additional problems. After completing a satisfactory compile, it is important that the backbone receive adequate structural and functional testing, avoiding any units which are outside the backbone.

Finally, once the backbone has been tested, additional modules are added and the entire program/system is again tested. This process will then continue until all the individual modules have been integrated, tested, and the program/system is ready for the system level phase of testing.

[Schach, 1999] compares the strengths and weakness of the integration testing techniques. He concludes that the best approach to integration testing is a sandwich implementation and integration approach which is able to assist in the isolation of faults early in the implementation. Major design faults are identified and components which have a probability of being reused in the system are properly tested thus preventing them from negatively affecting the system. Essentially

one is applying top-down integration for logic units and operational units are integrated bottom-up. This type of approach provides for the best of both worlds.

3.4.1.5 Data Flow Testing

[Harrold and Soffa, 1991] focus on another approach to integration testing by extending data flow testing to integration testing. The technique applies the concepts of data flow testing of modules to the larger issue of testing data flow between subroutines. Data flow testing is based on the analysis and identification of data dependencies, that is, where the variables are defined and where they are used. Data flow testing is extended to integration testing by creating an analysis technique plus a set of tools that can then be used to test the program.

In this approach, static analysis and program instrumentation are divided into two steps: 1) static analysis of the program to compute the inter-procedural definition-use pairs for the test case requirements, and 2) dynamic testing to determine whether user-supplied test cases meet the requirements.

Within the static analysis the program is first analyzed to identify all control and data flow information for each subroutine. This data is then fed to an analysis tool that creates an inter-procedural data flow graph. An analysis of the data flow graph follows to produce additional information regarding the actual nodes and edges of the graph. Here information is gleaned which relates to parameters which can reach across procedural boundaries. Finally, the definition-use data from step one and the inter-procedural data from step two are combined to create the requirements for the test cases. These test cases are then used to test the software. The tool that they developed for the analysis of the software and the creation of the inter-procedural data flow information is also used to instrument the program so that the program can be monitored during its execution.

The technique proposed by [Harrold and Soffa, 1991] is an important modification to traditional integration testing. Still, the same concepts are applied: location of the procedures and the related calls to those procedures, by either static or dynamic analysis, then instrumentation of the program to visualize its internal execution. The authors, however, do expand on the analysis and use of the data created during the analysis phase. In that way they are using a new method to perform integration testing.

3.4.2 System Testing

Integration testing and system testing are closely related. System testing is often viewed as acceptance testing. At that point of testing one compares the specifications with the resulting software. This phase of testing is often performed by a separate software testing group, or in certain instances, is carried out by the end-users. It is the focus of this research to look at system testing as the last phase of integration testing. System testing is divided into functional testing and structural testing. Structural testing relates to the program's internal structure, attempting to identify defects which are a result of incorrect coding of the software. Functional testing analyzes the operation of the program in comparison to the system specifications [Perry, 1991].

3.5 Summary

Integration testing is concerned with the relationships of various subroutines within a given program or system. The calling of subroutines, the passing of arguments, and the returning of results are important aspects of software. Both the functionality and the structure of the software are key components of acceptability. Integration testing is an important milestone on the way to system testing and implementation. Information generated during the integration phase is invaluable for final system testing and documentation.

Integration testing uses a combination of manual and automated techniques. Data gleaned from

the source code, as well as, data produced during program execution, are used to determine how the actual software performed, what data elements were used, which ones were modified, and how they were changed. Identification of the modules and their interfaces assists the tester in highlighting program weaknesses and faults. Furthermore, test suites can be developed based on the structural information produced by the integration analysis tools. Various analysis tools, such as, Call Tree Graphs (both static and dynamic), Data Flow Diagrams, Data Dictionaries, and Process Flow Graphs, are used to conduct integration tests. Comparisons between the system specifications and the actual implementation help one make a determination about the acceptability of the software system.

In the next chapter attention turns to integration testing of object-oriented software. Emphasis is placed on the types of tools and techniques that have been recommended for use with object-oriented software. A comparison is made between integration testing of procedurally-based software and testing of object-oriented software.

CHAPTER 4

Techniques for Object-Orientation

4.1 Introduction

Chapter 3 presented an overview of integration testing as traditionally applied to software. Emphasis was placed on the functionality of software with the procedure or function identified as the elemental unit of concern for integration and testing. Object-oriented software focuses its attention on both behavior and structure. Data and behavior are combined to form the object, the building block of object-oriented software. Thus, in object-oriented software the object, the instantiation of the class, is considered the focus of unit and integration testing.

This chapter examines the components of object-oriented software that impact on the ways in which software is tested. Attention is drawn to those fundamental concepts that cause one to use different techniques for carrying out software testing.

Existing research into the testing of object-oriented software is examined. An effort is made to examine the object-oriented model in a generalized manner. However, the implementation of the concepts within a particular language does impact on the use of these concepts, as well as, the actual testing of the software.

4.2 Implication of the Principles of Object-Orientation on Testing

One of the major reasons for object-oriented software is the improvement of programmer productivity through code reuse. By tying program behavior and data structure together in the form of classes and developing libraries of classes software can be constructed from classes that have been previously developed and tested. The object-oriented methods in software development include the concepts of encapsulation, inheritance, and polymorphism. Each of these concepts is discussed below, along with their impact on the testing of the resulting code.

4.2.1 Encapsulation

Encapsulation is an important element in the object-oriented model. Encapsulation relates to combining both the data and the methods used to access that data into a single unit, in C++ the class. Classes correspond to abstract data types (ADTs) with objects being the instantiation of these ADTs. Classes do not represent memory allocation and cannot be tested, for these reasons, the object is the focus of testing. To test a class one must first create an object of that class.

In C++ there is loose cohesion among the member functions. In many instances the individual member functions have only limited relationship to each other. The relationship of one member

function to another is often nonexistent except that they belong to the same class. Even though there is loose cohesion among the member functions, each of a class's member functions has access to all of its *public* and *private* data. Generally, one does follow the specification established for the individual member functions, thus preventing incorrect attribute access. However, due to the openness of access of public functions, it is possible to incorrectly allow access to an unrelated data attribute of the class. This fact influences the potential for defects being introduced into the software. For this reason, testing of object-oriented software must be more comprehensive.

Under C++, classes can contain components that are *public*, *private*, and *protected*. *Public* components are accessible by the user by merely addressing the name of the object, followed by a period (called the dot operator) and the name of the component being accessed. *Private* components have restricted access and generally can only be accessed by the members of the *Public* interface to the object. It is common practice to use a *Public* method or member function to access the *Private* data belonging to the object. *Protected* components are special instances of *Private* components of a class. *Protected* relates to derived classes. Elements declared as *Protected* can be accessed by objects of derived or child classes.

Encapsulation impacts on testing of software in a number of important ways. When performing unit testing one may have to exercise multiple functions in order to interact with the data attributes of the class. Completing unit testing correctly will in fact reduce the amount of time required for testing when reusing that code. Incorrect declaring of a class member can result in either restricting access improperly or granting access that results in incorrect modification of attributes. These defects may not be found during unit testing.

4.2.2 Inheritance

As discussed in Chapter 3, in traditional procedurally-based software the fundamental unit for testing is the subroutine. In C that unit is the *function*. To test a function one simply creates a

driver for the function, passes the required arguments to the function and evaluates the returned value(s).

Under C++ the entire situation changes. First, not all classes are instantiated. That means that not all of the classes are represented by objects. Certain classes are base classes, and through inheritance, are used to create more specialized child classes. These derived classes are then instantiated. Usually the base class is not directly tested.

Multiple inheritance allows for the creation of derived classes from more than one parent class. Such inheritance leads to increased complexity within the software.

Depending upon how the base classes are created there may or may not be *constructors* coded by the programmer. A *constructor* function is used to create the object instantiating the class. If the base class receives no arguments, a constructor need not be written; the default *constructor* will then be called at the time the object is created. In numerous cases, however, there will be a *constructor* for the base class. If the base class contains a parameterized *constructor* all derived, classes must contain *constructors.*

Specific syntax in C++ is provided for passing arguments to the base *constructors* from derived class *constructors.* Example 4.1 provides an overview of this concept.

```
#include <iostream.h>

class X{
    protected:
        int a;
    public:
        X(int i);
    };

class Y{
    protected:
        int b;
    public:
        Y(int i);
    };

// Z inherits both X and Y
  class Z: public X, public Y {
  public:
      Z(int x, int y);
      int makeab(void);
   };

X::X(int i)
    {
    a=i;
    }

Y::Y(int i)
    {
     b=i;
    }

Z::Z(int x, int y):X(x),Y(y)
    {
    cout << "Initializing \n";
    }

int Z::make_ab(void)
   {
   return a*b;
    }

main(void)
   {
   Z i(10,20);
  cout << i.make_ab();
   return 0;
    }
```

Example 4.1 Parameterized Constructors

Here class Z is derived from both class X and class Y. Both of the base classes contain parameterized *constructors*. When the *constructor* for Z is called, the *constructors* for X and Y are called with the necessary arguments passed on the appropriate parameter. The object that is created is of class Z.

Destructors are related to constructors in that they destroy objects after they are no longer needed. Destructors provide a means for garbage collection in object-oriented programs. C++ provides for the creation of both default and specialized destructors. In many instances one will need a destructor that is dynamically bound to an object. Virtual destructors make certain that the proper object is deleted correctly. Virtual destructors related to the polymorphic aspects of object-oriented programs. Proper utilization of destructors ensures that memory is released once the object is no longer needed.

Inheritance enables the programmer to build additional classes and objects based upon characteristics of other classes. Improper use of inheritance, particularly multiple inheritance, can produce excessive complexity that makes testing difficult at best. Message passing within the child class is equivalent to the linking of functions within a procedural language in that the logical order of calling is similar. Testing must be performed that can test all of the interfaces between the member functions.

C++ provides for the use of *friend* functions. A *friend* function provides access to the *private* members of a class. A friend function has access to all *private* and *protected* portions of a class to which it is a *friend*. *Friend* functions are declared as prototypes within the class to which it is being declared a friend. *Friend* functions can be used in assisting testing. They can also conceivably result in improper access to private areas of a class.

Classes can also be declared as friends of other classes. The *friend* class does not inherit the other class but is granted access to the *private* areas. This again may result in defects being indirectly incorporated into the software.

4.2.3 Polymorphism

Polymorphism in its more generic definition means *many forms*. In object-oriented programming polymorphism is a variation of the type definition which allows for individual items with common

names to take on different meanings. C++ provides for polymorphism through function and operator overloading, and dynamic binding. Multiple functions can have the same name but must be distinguishable by their formal parameters. These functions may or may not return the same type of value, but they cannot have the same type/number of formal parameters.

Polymorphism is supported in C++ through the use of virtual functions. The term *virtual* refers to the fact that derived classes can have different versions of the function which sharing a common name. Most procedural languages resolve function calls at compile time. With virtual functions, the binding of functions can be deferred until the code is executed. This deferred function resolution is known as late binding. In order for late binding to occur, a function must be declared identically to the base function, including the return value.

Operator overloading, another form of polymorphism, allows programs to use the same operator, for example the '+' sign, for different purposes. The operator applied to the expression is dependent upon the context of the expression. For example the '+' in 1 + 2 stands for integer addition while the '+' in 'Dog' + 'House' stands for string concatenation.

When testing object-oriented programs both of these forms of polymorphism add to the complexity of testing and maintaining software. Still, the advantage of using these techniques does aid in software development from existing libraries of routines. No longer is the programmer concerned with learning multiple function names that conceptually perform the same task but only use different data types. With operator overloading, ease in programming occurs because expressions can be stated logically by using operators that are used as one would state them in a natural language.

Polymorphism can either relate to static or dynamic binding. Static binding occurs at compile time. By default member functions are statically bound. Dynamic binding, occurring at runtime, allows for the proper function to be accessed based upon which object is being addressed.

[Jamsa, 1994] points out that true polymorphism only occurs when that function determination occurs during program execution. Still, it is important to recognize the ability of C++ to support both types of binding of function linkages.

Polymorphism impacts on testing in two different ways. Using the same name for multiple functions requires that the programmer use the right member function. Incorrect function selection will result in defects that may not be easily found. Overloading of operators opens the software up for incorrect operation. Use of operator overloading requires that the programmer carefully choose which operators to overload. Overloading operators so that they are not logical in their meaning, e.g., overloading the '-' to stand for concatenation, causes confusion in both coding and testing.

4.2.4 Generic Classes

Another implementation of polymorphism is allowed through the use of *templates.* Templates provide a means of incorporating generic programming into object-oriented software development. Here the programmer is able to devise a generic method which is used by the compiler to produce the actual code as it is needed within the resulting program. When the program is compiling, a call to the generic method with a previously unused data type causes additional code to be added to the resulting object code.

Templates provide a means of writing programs that use generic code sections. Libraries of routines, such as the Standard Template Library (STL), assist the programmer in creating new software which manipulates different objects. An example is the use of the linked list. A linked list of accounts for an accounting system would be manipulated in a standard manner as would a linked list of members for a club membership database. The only difference is the data being used, not the implementation of the linked list.

Testing of object-oriented programs which incorporate the use of templates or generic classes

requires that the executable code be examined in order to test the individual member functions, particularly if the library routines are pre-compiled and the source code is unavailable. Examining the source code only provides one with the generic template and the calls, not the implementation of the individual functions. Testing of code containing generic classes is difficult since it prevents one from being able to trace execution through the use of a debugger. The source code remains in its abstract, generic form and is not expanded when one needs to trace through the source code with the debugger.

The STL, mentioned above, provides a set of templates that relate to data structures and corresponding algorithms. This library, now included in the ISO/ANSI C++ standard, was designed to improve programmer productivity. It was not developed with an eye toward testing and defect trapping. Therefore, when using the STL it is important that one thoroughly test the resulting software, including any necessary defect detection/correction routines.

4.3 Existing Integration Testing Techniques for Object-Oriented Software

With the increased interest in the use of object-oriented methods for software development, attention has turned to the need for research into testing of object-oriented software. A number of different techniques and tools have been suggested for use in testing object-oriented software. These approaches provide insight into how other researchers and practitioners view the object-oriented method and its impact on testing of software. Existing testing techniques for object-oriented software can be classified as data-flow-based techniques, state-based techniques, state-based object interaction, as well as, techniques based on formal specification models. The following section exams a number of these approaches, primarily focusing on the more practical techniques and not on the more formal models. Each of the techniques presented here is considered to have made a contribution theory and practice of testing of object oriented software and systems.

4.3.1 Data Flow Analysis

Research of [Harrold, 1994] focuses on the use of data flow analysis for the testing of classes. Harrold's approach builds upon the hierarchical structure of classes and inheritance. Because derived classes contain elements of the base class, it seems reasonable that once the base class has been tested, only those attributes that are new or affected during inheritance need to be tested. Where possible the test suites developed for the base classes can be reused when testing the derived classes.

Harrold's technique is directed at the testing of classes. With this testing one can focus on intra-class testing, as well as, inter-class testing. In reality, one is testing the interaction between methods, both those belonging to the same class and those belonging to other classes. Because of the nature of class attributes, it is possible to use the same technique in either instance.

It is hoped that through this technique it is possible to reduce the amount of time required to perform testing. The primary effort is to perform appropriate tests for all interacting member functions of the same base class. Once this type of testing is completed then one can proceed to the integration of derived classes. The testing history for the parent class(es) is modified for use in testing the derived class(es). Any NEW or VIRTUAL-NEW member function must be thoroughly tested in ways defined in the base class [Harrold, 1994].

To implement this testing technique Harrold modified the GNU C++ compiler in order to output all identities as the parse tree is constructed. The output file contains all of the member functions and their interactions between all other member functions.

The second part of Harrold's technique focuses on subclass testing. Harrold uses data flow testing for this component. 'In data flow testing, subpaths are executed from a variable assignment (i.e., *definition*) to points where the variable's value is used (i.e., *use*)' [Harrold, 1989]. Harrold notes that data flow testing has proven effective in identifying defects. Again

Harrold modified the GNU C++ compiler to produce a list of definition-use pairs.

This technique is based upon static analysis of the source code. As mentioned earlier, the C++ use of templates, late binding, and other dynamic aspects preclude one from obtaining all of the necessary information from the source code. Still, this technique is helpful in developing greater insight into the actual software and for use in performing intra-class testing.

4.3.2 Jorgensen's Method-Message

The work of [Jorgensen and Erickson, 1994] emphasizes the specific issue of integration testing. Recognition is given to the differences between traditional/procedural programs and object-oriented programs. These differences require one to change the testing focus from structure to behavior. The lack of a *main* program and dynamic binding requires a new way of analyzing programs to be developed and a new set of rules for integration testing.

Jorgensen and Erickson present a conceptual means of viewing the execution of object-oriented programs. This view centers around the concept of the *Method/Message (MM) Path.* A MM-Path is a 'sequence of method executions linked by messages' [Jorgensen and Erickson, 1994.] The actual MM-Path commences with the execution of a method (in C++, a member function) and continues until a method is reached that does not 'issue any messages of its own'. According to Jorgensen, using such an analysis of an object-oriented system allows one to establish a set of test cases that can effectively perform integration testing. As with traditional testing, Jorgensen and Erickson utilize a bottom-up testing strategy. Beginning at the lowest level, the individual member function, testing progresses through five levels:

1) method testing,
2) message quiescence,
3) event quiescence,
4) thread integration, and

5) thread interaction testing.

One of the limitations of the MM Path method is the requirement that one must develop a set of tools capable of producing the MM Paths from the actual software. Again, software developed from class libraries may not avail itself to the automated creation of MM Path diagrams since certain MM paths may be hidden from the programmer due to the lack of available source code for analysis purposes.

4.3.3 Binder's FREE (Flattened Regular Expressions)

Binder sets three goals that must be met in order for software to be acceptable. These goals are: 1) each component must behave correctly, 2) collective behavior is correct, and 3) no incorrect behavior is produced. To achieve these goals, test cases must be developed that are directed toward unit, integration, and system level testing. Testing must be carried out that focuses on the identification of the most probable faults while at the same time being efficient and facilitating the creation of the test cases.

Robert Binder developed a conceptual framework in order to properly test object-oriented software. This framework can be used for the creation of test suites that are then used for both white and black box testing. The actual test cases are developed from the OOA/OOD models and are 'state-space confirmed' [Binder, 1995]. Test coverage is measured in regard to state coverage.

Binder's FREE is used to create specification-based test cases. Testing is handled at the unit, integration, and system levels. The testing goals for a specific system are used to impact the degree of testing carried out. FREE is not method dependent so it can be applied to any object-oriented software. As with Jorgensen's MM Paths, FREE is quite complex and requires that one create the testing models from the actual code.

Binder points out that state models are good for modeling classes/objects and the use of finite state machines is appropriate for the testing of object behavior. Binder developed his FREE approach to the testing of object-oriented software because '[s]tate-based testing provides a straight forward means to develop test suites that will reveal ... faults' [Binder, 1995].

The FREE approach addresses the following classes of tests: unit cluster, integration, and system. State based testing requires one to define states (variable values), expected output from member functions, and all legal and illegal sequences of method calling.

In order to utilize state-based testing, Binder provides the necessary elements to perform testing. These elements are:

1. State definition over a set of instance variable values limited to acceptable results of computations performed by a method.
2. Proper definition of transitions.
3. Events representing any action which cause a state change.
4. Actions representing output messages.
5. Use of special states used to aid in testing of constructors and destructors.

States are evaluated and the events that caused state changes are noted. Models are based on regular expressions. For Binder, inheritance needs to be considered for purposes of (re)testing. However, it does not affect the class under test; therefore, Binder assumes flattened classes.

The FREE approach extends black-box testing to cluster-testing of objects. Data-flow graphs are utilized for mapping the changes of states within a class. Extensions to these flow graphs are used for black-box and cluster analysis and test case design. The test suites are used to test the behavior of a class. In testing the behavior of a class one is looking for proper behavior, as well as, determining if there is any excluded behavior that should exist.

For integration testing Binder is concerned with the collaboration of a group of objects. He points out that the same technique can be used for both class and system level testing. Again, state information is used to develop test suites that properly test the interaction among the objects. The entire application can be viewed as a 'state machine network' and the combined states of all of the objects can be thought of as the current state of the system.

At the system level Binder acknowledges that, in order to perform testing, one must have a 'system-level specification' that can be used to develop appropriate scenarios. Binder sees the use of FREE at three different levels when considering system-level testing. These three levels are based on the completion of acceptable unit-level tests. Here Binder provided the criteria for completing various degrees of completeness in system testing.

4.3.4 Poston's Modifications to Rumbaugh's OMT

[Poston, 1994] examines the object-oriented paradigm and its relationship to testing. He points out that the Object Management Technique (OMT) life cycle of Rumbaugh is not adequate for testing since it relies on testing that occurs after one begins to implement objects. Poston concludes that object-oriented testing requires that test cases be developed in conjunction with design and development of the software. To overcome this inherent weakness in OMT, Poston enhances the life cycle by including test components at each phase of the development life cycle. This modification to OMT reinforces the arguments of [McGregor and Korson, 1994] in regard to the need for integrating testing throughout the entire life cycle.

Poston notes that OMT does not adequately provide for the inclusion of the object domain data, however, it does handle events, states, and state-changes. To resolve this weakness Poston enhances OMT by adding data domain notations to the objects themselves.

4.3.5 Firesmith's ADM3

[Firesmith, 1993], in discussing his Advanced Software Technology Specialists (ASTS) Development Method 3 (ADM3), focuses on the activities connected with sub-assembly integration and testing. Once the sub-assemblies have been integrated and tested it is necessary that these subassemblies be integrated and tested at the assembly level. The assembly continues to grow until the software is complete and system level testing is performed. This more advanced level of integration and testing consists of a number of different steps. These activities are composed of the following tasks:

1. Place the sub-assembly software into the subassembly library.
2. Plan sub-assembly integration and testing.
3. Examine the project-reuse repository for subassembly-level test software and test cases.
4. Design, code, and compile all of the sub-assembly-level test software and test cases not found in the project-reuse repository.
5. Integrate the objects, classes, and auxiliary units (if any) into a functioning assembly.
6. Perform the initial sub-assembly -integration testing.
7. Perform the peer-level assembly integration readiness inspection, to determine whether the sub-assembly is ready to turn over for integration with the growing assembly.
8. Place the sub-assembly software and documentation under developer configuration control.

[Firesmith, 1993]

One should note that Firesmith's technique focuses on both testing and documentation. The documentation assists in project management, as well as, providing for test case reuse.

Since object-oriented software development focuses on the concepts of reuse, it is appropriate that the testing phase also consider reuse by building repositories of appropriate test suites. The

focus of Firesmith's method is on both reuse of existing test cases and test software, and the proper documentation for use in both project management and accountability. There is limited focus on how these steps will add to the quality of the resulting software product.

4.3.6 Tai and Daniels Interclass Testing

In a related research project [Tai and Daniels, 1999] examine interclass testing. They recognize that integration testing for object-oriented software includes testing of both intra-object communication between modules belonging to the same class and communication between objects of different classes. Their emphasis is not on the actual testing of the various objects but on the order in which they are tested. Tai and Daniels recognize that traditional integration testing, based on the "call" relationship between objects, does not satisfy the need for integration testing of object-oriented software because 1) "methods in a class often interact with each other through shared variables" and 2) "the call relationship is not the only type of relationship between objects" [Tai and Daniels, 1999].

Their method contains two primary parts: 1) " assignment of level numbers to classes in an ORD [object relation diagram]" and 2) "integration testing based on levels numbers of the classes" [Tai and Daniels, 1999]. The values are assigned to the various classes based upon the association of the various classes as demonstrated though the construction of the ORD. By their method of assigning major and minor numbers to the classes in a program one can determine what order to follow in testing the software. The use of drivers and stubs for testing are identified by the use of these numbers.

Tai and Daniels argue that if one follows their method that one can satisfy the properties necessary for proper interclass testing. They do point out that the actual test cases necessary for the interclass testing is not part of their research. Furthermore, the effect of polymorphism and dynamic binding must be studied as it relates to interclass integration testing.

4.3.7 Coupling Based Testing for Polymorphic Relations

[Alexander, 1999] proposes the use of coupling relationships for integration testing based upon the work of [In and Afoot, 1998]. He extends their initial work to "accommodate the object-oriented-oriented features of inheritance and polymorphism" [Alexander, 1999]. Alexander argues that there is insufficient research being performed on integration testing of object-oriented software. In order to resolve this weakness in object-oriented software development, he proposes research be carried out on component coupling.

Alexander identifies twelve different coupling cases within object-oriented systems. Each of these cases is then evaluated in regard to the effectiveness of using coupling-based testing criteria. In order to evaluate his integration testing technique, Alexander suggests the development of a tool that takes JAVA as its input, instruments the code, and the, with the use of a test driver, executes the instrumented code, producing results which are processed by a results analyzer.

Alexander believes that the results of his research will be as follows:

> 1. An effective way to do integration testing of object-oriented programs.
> 2. An extended set of coupling test definitions sufficient for testing object-oriented programs.
> 3. A set of test adequacy criteria based on the extended coupling definitions.
> 4. A taxonomy of faults that result from the use of inheritance and polymorphic behavior in object-oriented software.
> 5. A set of metrics for measuring software based on couplings.
>
> [Alexander, 1999]

At present, the results of Alexander's research has not been published; however, the fact that he is focusing on object-oriented software integration testing, it is important that his work be noted. Additional work on this topic is presented in [Alexander and Offutt, 1999].

4.3.8 State-Based Testing

To test object-oriented software, state-based testing was examined as a possible solution. State-based testing emphasizes the use of state diagrams to determine the proper states an object can exist in under different conditions. As objects interact during the life of the program their states will change as they communicate with other objects. One of the primary research efforts concerned with state-based testing of object-oriented software is presented by Turner and Robson.

4.3.8.1 Turner and Robson

[Turner and Robson, 1993] focus on unit and integration testing of object-oriented software and the lack of research in this area. They identify state-based testing as a key component of testing object-oriented software. Traditional testing of procedurally-based programs has often utilized data-flow analysis and has been functionally oriented. With the object-oriented paradigm there is need for examining both the behavior and the structure of the software. In order to properly carry out such analysis/testing, Turner and Robson consider state-based testing as the proper type of testing for object-oriented software. State-based testing examines the states of variables before and after functions are called. With object-oriented software one is concerned with the state of an object. The state of an object is defined as the combined state of all variables contained within that object. These states can then be compared to the known states that should exist. The known states are determined from the system design. To determine the state of an object one must know its current state and then, through the use of a Finite State Automata, determine what state the variable(s) will be in after execution of a set of statements. Within object-oriented software there will be member functions that will change the state of the object and member functions that only inquire into the object's current state. The states that can result from the call of a member function are:

1) The object's state can be changed to an appropriate new state.
2) The object's state can remain as it is.

To test object-oriented software, state-based testing was examined as a possible solution. State-based testing emphasizes the use of state diagrams to determine the proper states an object can exist in under different conditions. As objects interact during the life of the program their states will change as they communicate with other objects. One of the primary research efforts concerned with state-based testing of object-oriented software is presented by Turner and Robson.

4.3.8.1 Turner and Robson

[Turner and Robson, 1993] focus on unit and integration testing of object-oriented software and the lack of research in this area. They identify state-based testing as a key component of testing object-oriented software. Traditional testing of procedurally-based programs has often utilized data-flow analysis and has been functionally oriented. With the object-oriented paradigm there is need for examining both the behavior and the structure of the software. In order to properly carry out such analysis/testing, Turner and Robson consider state-based testing as the proper type of testing for object-oriented software. State-based testing examines the states of variables before and after functions are called. With object-oriented software one is concerned with the state of an object. The state of an object is defined as the combined state of all variables contained within that object. These states can then be compared to the known states that should exist. The known states are determined from the system design. To determine the state of an object one must know its current state and then, through the use of a Finite State Automata, determine what state the variable(s) will be in after execution of a set of statements. Within object-oriented software there will be member functions that will change the state of the object and member functions that only inquire into the object's current state. The states that can result from the call of a member function are:

1) The object's state can be changed to an appropriate new state.
2) The object's state can remain as it is.

3) The object can be placed in an undefined state.
4) The object can be placed in an inappropriate state. [Turner and Robson, 1993]

States 2, 3 and 4 may be faults. At least the last two states for an object are incorrect. Still without prior knowledge of appropriate states it is impossible to know if the software is indeed behaving incorrectly.

Turner and Robson recognized that the state of an object can change over time. For this reason they established a new definition for the state of an object - 'the combination of all of the sub-states of the object at a point in time' [Turner and Robson, 1993]. With this definition they divided sub-states into two different types:

1) Specification sub-state values.
2) General sub-state values.

To determine the important sub-states one must perform the following steps for each data member of the object:

1) Analyze which particular values are significant (by analysis of the code, or the design) and which are not.
2) Allocate one sub-state value for each significant value.
3) Allocate one sub-state value for each group of related values.

To facilitate the collection and analysis of object states one must modify the object under test. The simplest way to carry out state reporting is to include output statements in each method and then report to either a file or to the screen the values of arguments passed to and values returned by the member functions. Turner and Robson also see this technique as a means of examining sub-state values that may be helpful in debugging and execution tracing. An additional extension to the collection of data about object states is the inclusion of assertions before each function is

called. The use of assertions is a technique that allows the software to test itself, confirming that things are what they are supposed to be. If an assertion is included in the code an defect will be reported if a condition exists, at the time the assertion macro is executed, that violates the assertion statement. If one utilizes the assertions in C/C++ then the defect is reported and the program either terminates or reports the defect. In the C language assertions are implemented as macros. In C++ assertions can be implemented by the use of templates. Stroustrup discusses the concept of invariants as 'a piece of code that can be run to check the state of an object' [Stroustrup, 1991]. Invariants are usually not left in the final software but are used during the testing and debugging phases.

In order to know the states and sub-states of an object Turner and Robson use a technique, data scenarios, that allows one to define the specific states. They point out that one should focus on the underlying data structure(s) that is used for the particular feature of the object. Their tool, MKTC, allows for the setting of the necessary start value and then uses data scenarios, instead of state descriptions, to trace the various object states and substates. Turner and Robson provide a series of steps to be followed in generating the data scenarios, that is, the generation of the test cases:

1) Allocate one sub-state per data member of the class under test.
2) Determine the data scenarios from the design of the class.
3) Allocate the extra sub-states required for the data scenarios to function properly.
4) Determine the specific values and the general values for all these sub-states.
5) Add the features to test for the sub-state values to the class.
6) Determine which sub-states require a change of value test.
7) Analyze from the design the call graph for inter-features within the same class.
8) Start with features at the bottom of the graph.
9) Generate the code to create the test case scenarios, that is, the starting state of the object. Add code for the test, including calls to the feature under test. Apply a test for the final state of the object and any code that is required to clean up after

the test.

10). Repeat steps until there are no other features to test.

[Turner and Robson, 1993]

The weakness of the technique relates to its need for dynamic classes and methods. If the class under test is merely a repository for data and the data structure is non-dynamic, then their technique has limited usefulness. The more the features of a class interact with its data representation, the more effective state-based testing is. Turner and Robson recognize that state-based testing is not a substitute for functional or structural testing, but that it contributes to the testing of object-oriented software.

4.4 Summary and Conclusions

Existing research into testing of object-oriented software has focused on the extension of the system design models (e.g., Poston), the use of dataflow analysis (e.g., Harrold), and the use of state-based techniques (e.g., Jorgensen, Binder, Turner, Robson). Each of these techniques brings insight into object-oriented software, but each possesses it own weaknesses. Additional work has been carried out that examines the use of formal specification languages. [Liu, 1996] finds that if one makes modifications to dataflow analysis to handle system level data, it may be an acceptable means of testing object-oriented software.

From the work discussed in this chapter it is possible to conclude that object-oriented software brings complexity into the testing arena. The question of testing object-oriented software has been examined and it has been determined that there is a uniqueness in object-oriented software which calls for additional software testing techniques.

As one can see from the discussion presented in this chapter, integration testing of object-oriented software is an important topic that has taken on additional relevance due to the complexity

brought on through the use of polymorphism, reusable class libraries, templates and related object-oriented concepts. Chapter 5 explores an approach to integration testing that employs the use of post-implementation and design models in support of integration testing.

CHAPTER 5

Integrating Views for Testing

CONTENTS

5.1 Introduction

Traditional integration and testing of software focus on the units of which software is composed and the interaction between these units. The makeup and nature of object-oriented software changes both the ways in which software is integrated and the ways in which software should be tested. Integration is now concerned with the interaction between objects and groups of objects viewed as components. Since much object-oriented software is event driven, it is unlikely that one can identify all of the sequences of interactions. For this reason the internal architecture of the software has to be examined both statically and dynamically in order to develop proper test cases and to identify defects and areas within the software suspected of faults. This chapter examines the need for testing object-oriented software in a new manner. The same diagrams and models used in the analysis and design of the object-oriented software are recommended for use in the evaluation and in assisting in the testing of the software during the implementation phase.

In this chapter the Unified Modeling Language (UML) is examined as it is suggested that UML be used for developing the models employed during the testing phase. UML is becoming an accepted standard for object-oriented analysis and design modeling. The suitability of UML as a modeling language for developing static and dynamic models during the software development life cycle is appraised. The rationale for creating post-implementation models of the software is presented as these models support the testing of software. The comparison of the post-implementation models with the analysis and design models is proposed as a mechanism for assisting in integration testing and for identifying the presence of defects prior to actually performing integration testing. Elimination of certain of these defects can improve the effectiveness and efficiency of integration testing of object-oriented software.

5.2 Object-Oriented Integration Testing

Chapter 3 presents a thorough overview of integration testing of procedurally-based software. The emphasis of traditional testing is on the functional composition of the software. Subroutines

are developed and then tested as units. As the subroutines are released to the build, integration testing is performed that evaluates the calling sequence, the passing of arguments to the subroutine, and the returning of values. In procedurally-based software the flow of execution is often observable from the source code. The goals of integration testing are discussed in Chapter 3.

Integration testing of object-oriented software requires both the testing to code and the testing to specification. One must examine both the implementation, as well as the design since many aspects of the implementation are hidden through the use of templates, inheritance, and dynamic binding. Many of these aspects are the result of the language, particularly C++, chosen for implementing the design.

Object-oriented software differs from procedurally-based software in that it tends to be event driven, and thereby, less traceable from the source code. Enhancements in object-oriented languages, especially C++, have added features that support both reuse and generalization. These concepts, discussed in Chapter 4, create the need for rethinking software testing. Consideration must be given to the nature and construction of object-oriented software. Traditional software testing techniques must be modified for use in testing object-oriented software and the languages in which it is built. To meet the need for appropriate techniques and methods for testing it is important that one understand both the process of creating and the resulting structure of object-oriented software.

For object-oriented software, integration testing can be defined as the testing of the interaction of units of which object-oriented software is constructed. Units are defined as objects, clusters of related objects, subprograms, or individual programs. Integration testing can be expanded to include both the software and the representation of the different models of the software, physical, logical, and functional, and how they relate. Specifically, one is concerned with the relationship of not only the components of which the software is composed but also the connectivity of the

design and the implementation as expressed in the documentation and models.

Internal testing of classes is considered sub-unit testing. This type of testing is closely related to white-box testing in procedurally-based languages. In object-oriented software, it tests the internal construction of methods belonging to a class. The interaction between member functions, however, may need to be considered when discussing integration testing of object-oriented software. Traditional integration testing focuses on the interaction among units, usually functions or procedures, and examines the external interface of these units, i.e., arguments passed and values returned. In object-oriented software written in C++, it is common for an object to send a message to itself via one of its methods. This type of interaction within an object requires testing.

To further complicate the issue of integration testing of object-oriented programs, objects can pass messages to other objects belonging to the same class or to a different class. The testing of objects belonging to the same class requires the same level of integration testing as the testing of objects from different classes.

Alternative methods of testing object-oriented software are presented in Chapter 4. Emphasis is placed on the aspects of the object-oriented paradigm that affect software integration and integration testing. Techniques for integration testing proposed by Harrold, et al., Poston, Jorgensen, Binder, Firesmith, and Turner/Robson are discussed.

In order to properly test object-oriented software one must understand both the process and the resulting software in order to determine when tests should be performed, what tests to perform, and what the tests reveal. Since this thesis is concerned with integration testing the focus of the remainder of this chapter is on this sub-phase of the implementation cycle. However, it is important to recognize the need for rethinking all aspects of software testing, from unit to system testing.

5.3 Integration Testing within the Software Development Life Cycle

To develop a technique for integration testing of object-oriented software, it is necessary to have an understanding where software integration and integration testing lie in the software development life cycle. Since the Revised Spiral Model for Object-Oriented Software Development has been chosen for use in this thesis, the discussion of integration testing's place will be associated with that life cycle.

In the Revised Spiral model various types of tests are performed throughout the life cycle. As software is developed unit testing occurs. Once the unit has been tested, it is released for inclusion in the next build. For this thesis the build is defined to mean anytime a new piece of code is added, not necessarily when the software is released as a usable system. This definition expands the common usage of the term build. At this point integration testing begins. Object-oriented software integration is far more complex than traditional procedurally-based software. Interaction of objects can change over time. Therefore, integration testing must continue throughout the development cycle until the software is fully integrated and ready for system testing. For this reason integration testing will often include objects that have already experienced integration testing. This characteristic of object-oriented software is supported by the Revised Spiral Model.

The Revised Spiral Model, illustrated in Figure 2.2, supports the entire development life cycle. Of particular interest are the 4th and 5th phases where software development and implementation occur. In phase four, coding of the software begins and object testing is initiated. Coding and testing continues at this level, while at the same time, integration testing begins once communicating objects are included in the build. Integration testing matures over the development life cycle. At first integration testing is concerned with object interaction, as illustrated in Figure 5.1, where Object A and Object B communicate with each other through message passing.

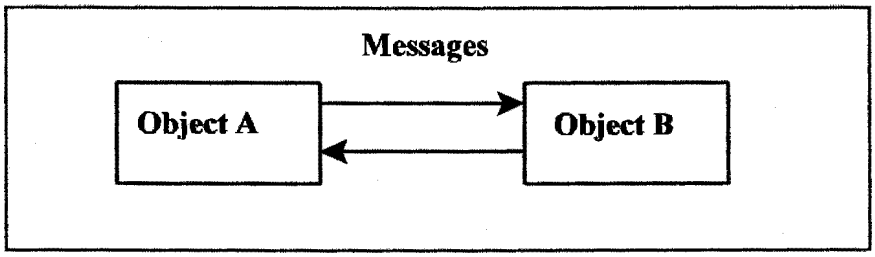


Figure 5.1 Object Level Integration

One is concerned with the object called, the message content (e.g. values passed) and the message returned. Sequencing and timing of messages expands the concerns of integration testing.

Eventually similar objects are grouped into larger program modules or components. The interaction between these modules may be restricted to a limited set of objects. Here integration testing examines the message passing between components, as shown in Figure 5.2.

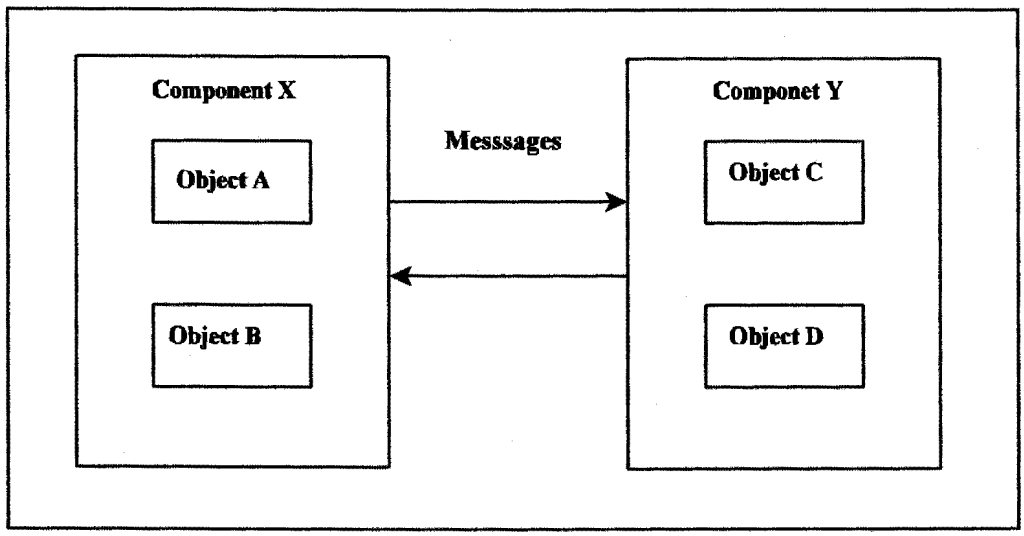


Figure 5.2 Component Level Integration

The Revised Spiral Model, as presented by du Plessis and van der Walt, does not present detail regarding object testing and verification. Since this thesis is concerned with integration testing of object-oriented software, the Implementation Cycle of the Revised Spiral Model is elaborated to show detail for unit testing and integration testing.

The 4th and 5th phases of the Revised Spiral Model can be modeled using a slightly different view that reflects the iterative nature of object-oriented software development. Testing follows coding and the cycle repeats until the entire project is completed. Figure 5.3 shows the iterative process of development as one codes objects, tests them, releases them to the next build for integrating into the software, performs integration testing, and then returns to more coding. It is assumed that whenever defects, are found the defect is removed and the phase continues. This iterative process continues throughout the development cycle.

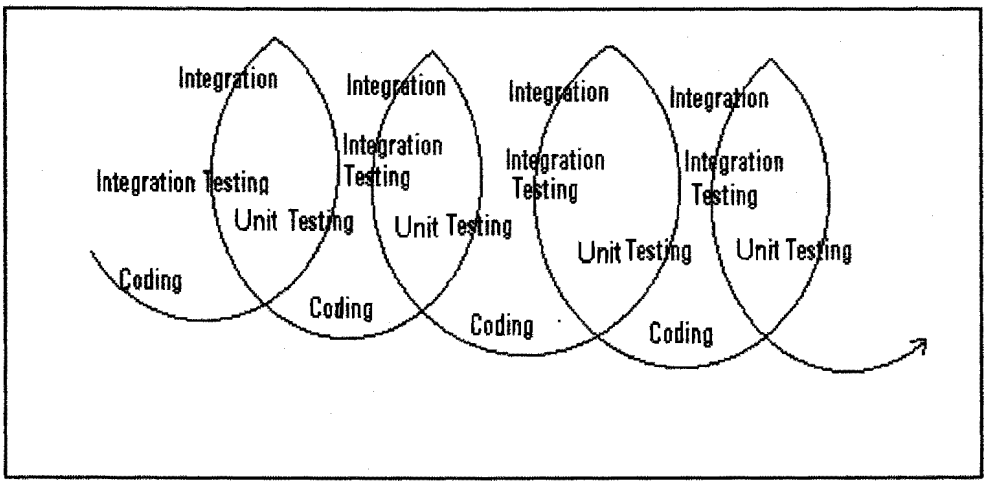


Figure 5.3 Iterative Aspect of Object-Oriented Development - Code, Test, Integrate, Test, Code...

Integration testing should occur at two key points:

1) once the object has been tested and is ready for inclusion into the software, and
2) when clusters of related objects have been tested and are ready to be integrated as sub-systems.

Integration testing is repeated at these points in the software development life cycle until the entire software system has been integrated and tested. At that point attention shifts to system testing. Figures 5.4 and Figure 5.5 show the addition of detail to the Revised Spiral Model to address two levels of integration testing. The terminology utilized in these diagrams is derived from the work of [Humphrey, 1989] as discussed in Section 2.7.1. In this thesis, both the Worldly and Atomic level models of integration testing are important to understanding when and how testing is to be employed.

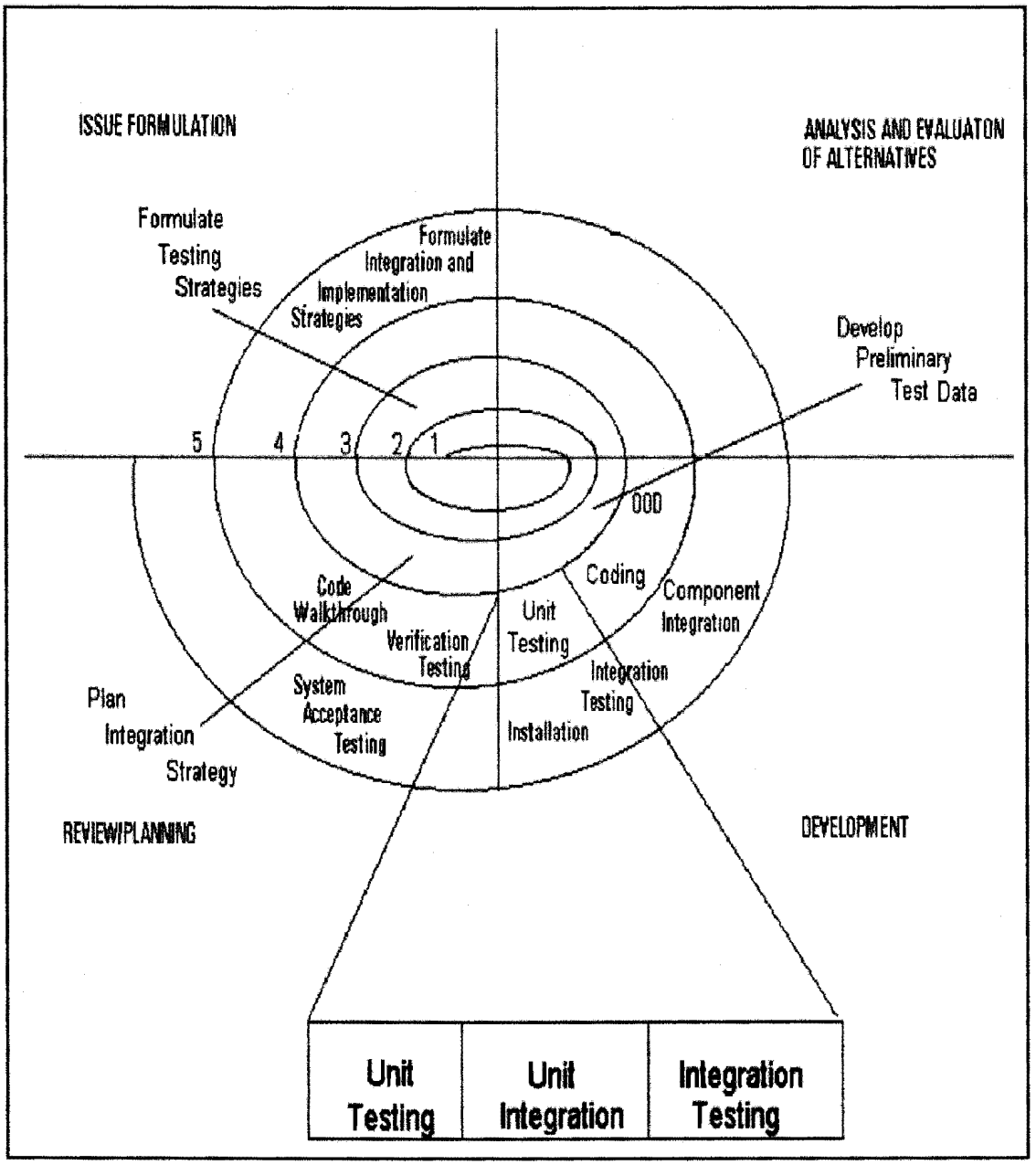


Figure 5.4 Worldly Level Implementation with Atomic Level Elaboration of Unit Testing

The exploded view of the Revised Spiral Model shows where unit testing will occur and the fact that integration testing occurs once a unit is released to the next build. Figure 5.5 illustrates the

next cycle within the Software Development Life Cycle where integration testing occurs. At that point one is concerned with the integration of components or subprograms.

As can be seen by this diagram, the Universal Level equates to the Revised Spiral Model which illustrates all of the elements of which the development process is composed. In order to further understand how the units are developed, tested, and then integrated, it is necessary to expand on the information presented in the Revised Spiral Model, by including such detail that is considered to be at the Atomic Level of a process model.

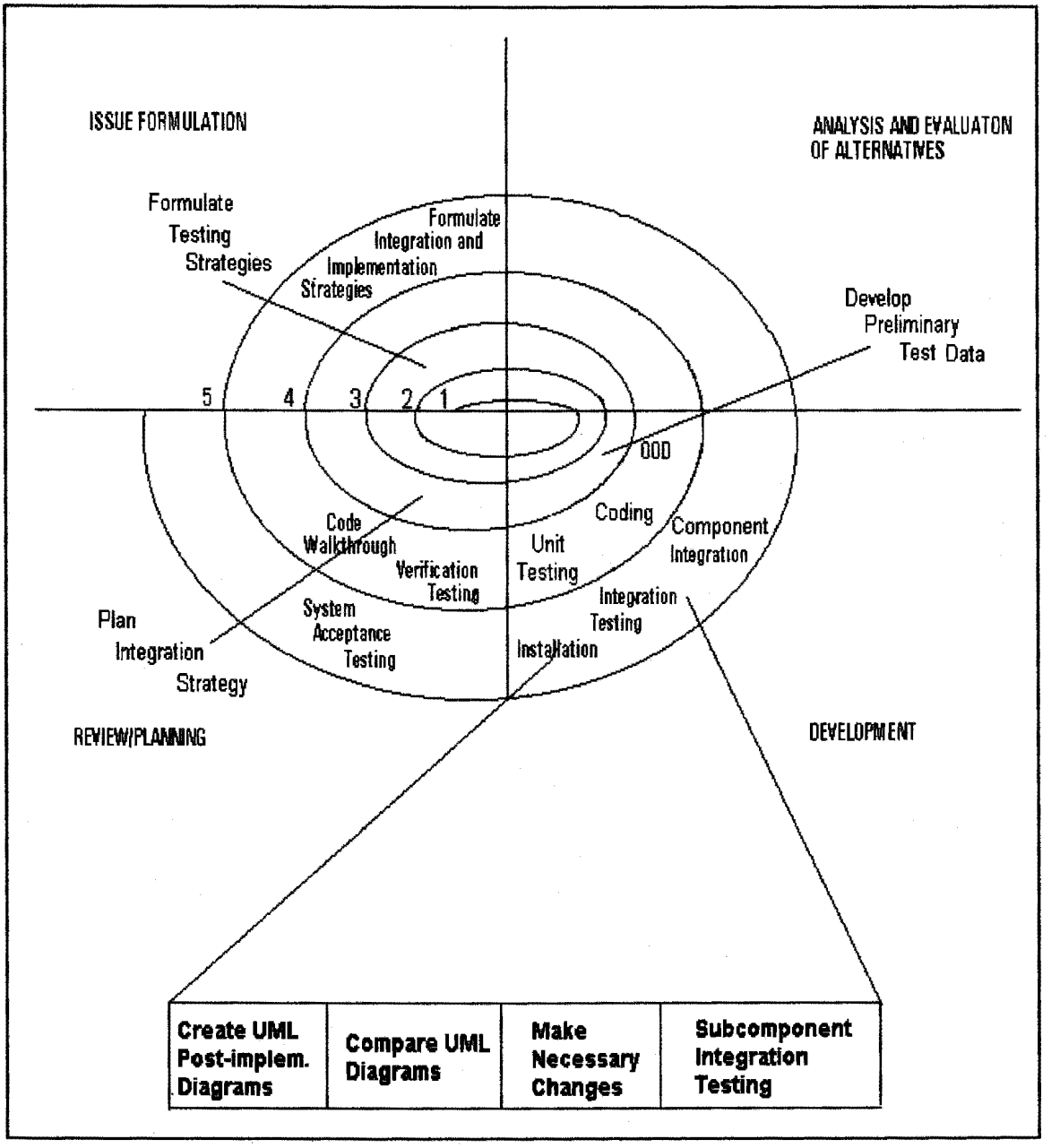


Figure 5.5 Worldly Level Implementation Cycle with Atomic Level Elaboration of Integration Testing

Note that integration testing is occurring shortly after object testing has been completed and again

after sub-system integration. In the first instance one is examining inter-object communication and in the second case, inter-component communication. Software testing can be viewed in the same levels of abstraction as the overall software development life cycle, as shown in Table 2.8.

5.4 Using UML to Model Object-Oriented Software

UML was selected for use in this thesis due to it being an accepted modeling language that is readily applicable to object-oriented software development, while at the same time being process free. UML provides the user with a means of visualizing both the logical and physical models of a software system. Utilizing UML, analysts and designers are able to communicate with both end users and programmers. Overall, UML is a very acceptable modeling language for use in developing object-oriented systems.

A model provides a means by which an abstraction can be illustrated which represents the essential structure of elements. The use of a standardized modeling language provides the entire software development group with a means of standardizing the analysis, design, and implementation documentation. UML was designed to provide a common interface to object-oriented software development. Through the collaborative effort of Booch, Rumbaugh, and Jacobson, UML was created in an effort to replace Booch's clouds, Rumbaugh's OMT, and Jacobson's Use Cases, as well as, other techniques and methods used for object-oriented software development; not by doing away with them, but by combining them into a modeling language that was essentially process free. This modeling language could then be used to provide visualization of the analysis and design of object-oriented systems. It was desired that through the use of such a modeling language, individuals involved in software development projects would be able to understand the designs, regardless of the software development process or object-oriented programming language being employed. This standardization of modeling and the unification of the approaches of Booch, Rumbaugh and Jacobson is considered to be a major step forward in

the institutionalization of the object-oriented paradigm.

UML provides support for modeling both the logical and physical views of object-oriented software. The logical view of a system represents the abstractions and mechanisms that comprise the problem domain or the system architecture [Booch, 1994] and is developed during the Analysis Cycle. The physical view presents the actual design of the software or the hardware components that make up the system the main deliverable of the Design Cycle. Each of these views can be modeled by using diagrams that represent either the static or dynamic aspect of a system.

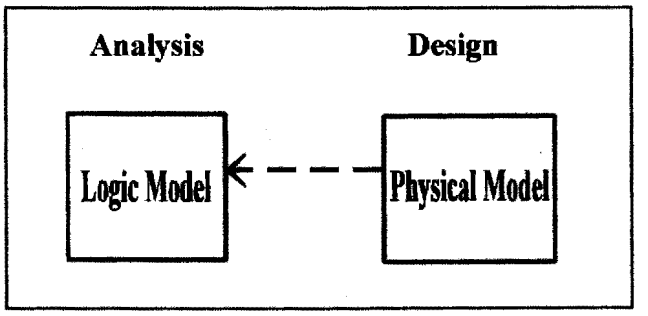


Figure 5.6 Dependency of the Physical Model on the Logic Model

The logical model is associated with the physical model in that the logical model is used to illustrate how the system is designed to solve a particular problem. The logical model represents the external view of the system. Here, in Figure 5.6, one sees the relationship and the dependency of the Physical Model on the Logical.

The physical model, on the other hand, represents the architecture of the system to be implemented. The physical model is derived from the logical model by adding physical attributes and structures that are necessary to construct the executable system. The link between the logical and physical models are a set of interfaces between the building blocks that form the architecture.

The physical model, in meta terms, is derived from the logical model. Collaboration, sequence, activity, state, and component diagrams are used to model the physical views of a system. To understand how the individual diagrams of UML are associated, the diagrams available in UML and the application of each to modeling object-oriented software are presented. In using UML for object-oriented software development it is essential that one understand not only the specific diagrams available and the syntax/semantics of the language, but the context in which these models exist. The actual syntax of UML is presented in Appendix A of this thesis. The following metamodels illustrate UML. UML is applicable at the Universal, Worldly, and Atomic design levels. UML can be used to represent the software under development at each of these three levels.

The physical, logical, and functional views, needed to understand a system's structure and behavior, are represented as interactions between elements of a system. These elements can be classes, objects, packages, sub-programs, components, and individual programs. Closely related classes can be combined into packages to assist in the management of large and complex system designs. Packages have meaning only during the analysis and design of the software and are not necessarily translated into the actual software. Components, on the other hand, represent parts of a software system. A component can be source code, object code, or executable code. In UML components often represent subprograms or sub-modules of a program or software system. This thesis often refers to components in this sense.

The different diagrams provided in UML, see Table 5.1, are used to build a comprehensive model of a system. These diagrams provide the user with a means of representing the different aspects of software being designed.

Table 5.1 UML Diagram and View Association

Structural View Models	
Class	Static structure of the system, logical view
Object	Implementation view of classes
Component	Division of software into cohesive subprograms during implementation
Behavioral View Models	
Use-Case	External user view of functionality
State	Possible states an object of a class may have and events causing state to change
Sequence	Interaction between objects in completing a task over time
Collaboration	Completion of a task by objects illustrating message passing and relationship of objects
Activity	Sequential control flow within a program or a member function

Booch, Rumbaugh and Jacobson present a slightly different classification of the UML diagrams and the views they represent [Rumbaugh, et al., 1999]. The major difference is the inclusion of the use-case diagrams under the header of Structural View Models. In this particular table emphasis is placed on the difference between diagrams representing static structure and program behavior.

In developing object-oriented software there are a number of different views, or perspectives, that one must have of the system under development. Software can be seen from the design and

the implementation views. Both the design and the implementation contain static and dynamic aspects. To further clarify this issue, Table 5.2 illustrates which diagrams are used to represent the static and dynamic models of the design and the implementation.

Table 5.2 Design and Implementation Views

Design	Implementation
Static class, object, package diagrams	Static class, object, package diagrams
Dynamic use case, sequence, collaboration, activity, state transition diagrams	Dynamic sequence, collaboration, activity, state transition diagrams

As is illustrated in Table 5.2, there is a correlation between the diagrams utilized in the design and those created from the implementation. It is noted, however, that Use Case diagrams can not be recreated from the actual implementation.

Combining the different diagrams allows one to understand the integrated nature of UML diagrams and the model's relationship to the logical and physical views of an object-oriented system. All of these four views, logical/physical and static/dynamic, are important in designing and implementing software. UML supports all of these views. Section 5.4.1 discusses in further detail the interrelated nature of UML.

These diagrams, used in a coordinated manner, provide the analyst and designer with a standardized modeling language for object-oriented software. Once the design has been documented in UML it can then be given to the programming staff. Each of the different

diagrams represents a different view of the system under development. These views are necessary for the actual coding of the software. In addition to the programming phase of the software development life cycle, the pre-implementation diagrams are helpful in the analysis and testing of the actual software. Section 5.6 discusses how these diagrams, along with post-implementation diagrams, derived from the software, can be used for identifying defects in the design and/or implementation and for the creation of test cases for integration testing of the software. As the design diagrams are important in the development of the software, the same holds true for software testing where diagrams representing the implementation view of the software can assist in the development of test cases.

5.4.1 UML Interrelationships

As discussed in Section 5.4, UML provides a number of different diagrams that are useful in modeling object-oriented software. Individual UML diagrams, however, cannot stand alone if one is to create a comprehensive model of the software. The recognition of the interconnection of these models is necessary if one is to have an understanding of UML, and how UML is used to model the different views of a software system. A macro-view of UML allows one to picture how various diagrams are related and their individual importance in the design of object-oriented software. Figure 5.12 presents the integrated nature of its various diagrams. This integrated perspective of UML is directly related to the integration technique presented in this thesis. The integration testing technique utilizes this integrated view in understanding the structure and behavior of the software under question. Test cases are also developed for integration testing by examining the detail design and implementation diagrams.

To understand the relationship of the logical model and the physical model, the different UML diagrams need to be individually examined to show their relationship with other UML diagrams. This leads to an understanding of the integrated nature of UML.

UML provides for modeling both the static and the dynamic views of a system, see Section 5.4. In developing software one often begins by first gaining an understanding of the external usage of the system. In object-oriented analysis use cases are identified and scenarios are developed to represent them. The use cases are then employed in providing information necessary for developing the internal structure and behavior necessary to implement those use cases. In UML collaboration and sequence diagrams are used to model internal system behavior at the use case level.

Class diagrams are the foundation of object-oriented software development. They provide the necessary information about data attributes and class methods. Once objects are identified during analysis, classes are designed to support those objects; when necessary class diagrams are used to assist in creating activity diagrams. Activity diagrams represent the internal control structure of an individual method. Activity diagrams are normally drawn whenever there is a unique algorithm or excessive complexity in a method.

After use cases have been identified, scenarios are developed that elaborate the sequence of actions depicting the behavior of the use case. Simply said, scenarios are instantiations of use cases much like objects are instantiations of classes.. From these scenarios objects are identified. Classes are developed to support each of the objects identified in the scenarios, see Figure 5.7. One takes the information contained in the class diagrams, e.g., class name, data attributes, and methods, in order to properly build sequence and collaboration diagrams. From this information one begins to recognize how UML diagrams are related to object-oriented development and to each other.

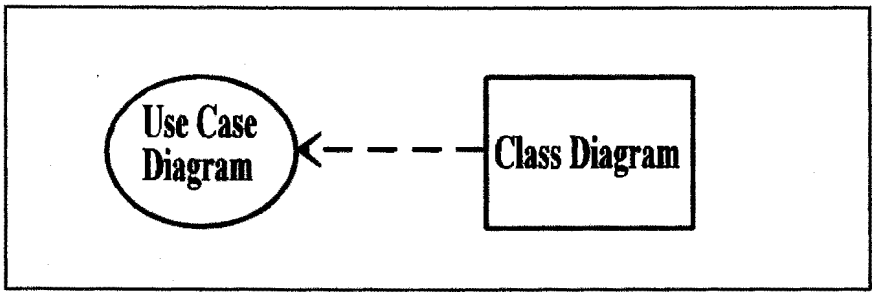


Figure 5.7 Use Case Diagram's Relationship to Class Diagram

As the classes are developed from the use cases then class relationships: inheritance, association, dependency, are identified. Groups of related classes can be combined into packages in UML as illustrated in Figure 5.8. Related packages can be reduced to a single package if it is necessary to manage the size of an object-oriented system design.

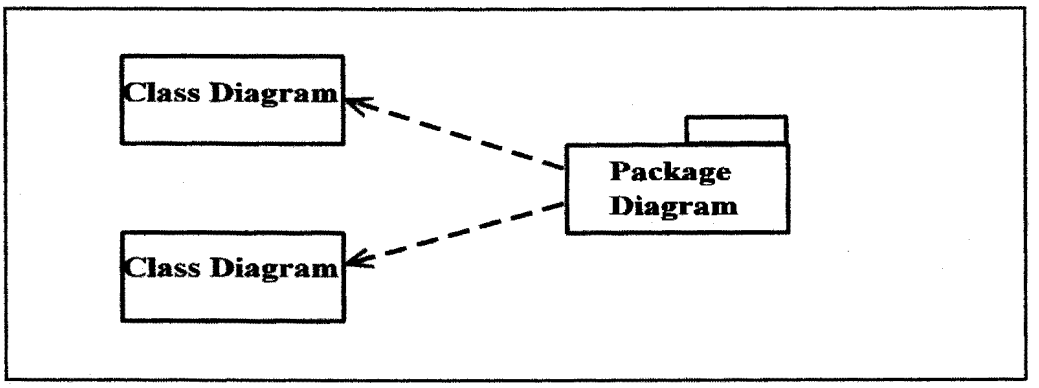


Figure 5.8 Package Diagram's Relationship to Class Diagrams

Once classes and their relationships have been identified emphasis is directed toward the behavioral and functional models. Interaction diagrams are used in UML to illustrate the passage of messages between objects. Sequence and collaboration diagrams model the internal

architecture of the system. All of these diagrams are used to model use cases.

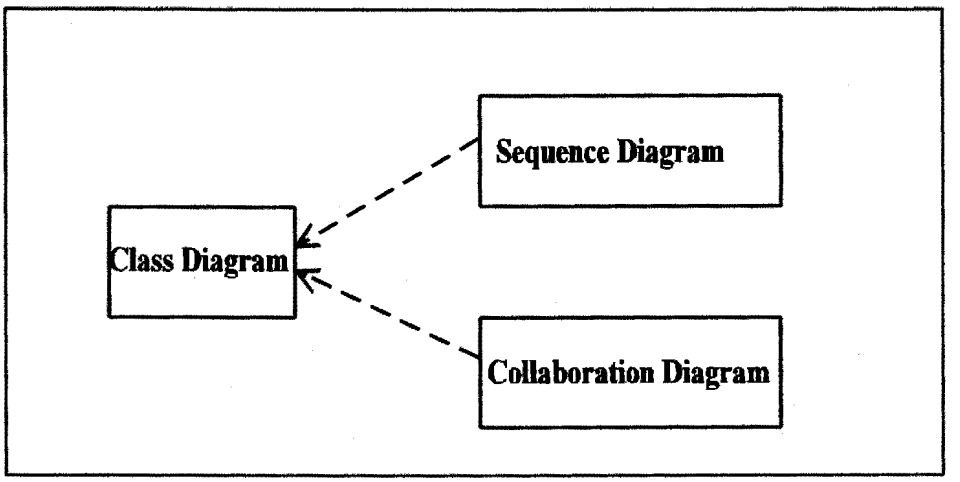


Figure 5.9 Class Diagram's Relationship to Sequence and Collaboration Diagrams

Sequence diagrams model the behavior of objects created from classes. Objects interact by sending messages to other objects using pre-defined methods. Sequence diagrams illustrate a sequential flow of events that are followed in order to complete a task. Sequence diagrams are primarily concerned with object interaction over time; whereas, collaboration diagrams contain classification and association roles. Collaboration diagrams describe the "configuration of objects and links that may occur when an instance of the collaboration is executed" [Rumbaugh, et al., 1999]. Collaboration diagrams model object behavior presenting a view of objects as they work together to complete a task. Figure 5.9 illustrates the relationship of Collaboration and Sequence diagrams to the Class Diagram.

A state transition diagram or statechart presents the internal behavior of a single object showing the states of the object, events (messages) that can cause transition in the object's state, and the actions resulting from a state change [Quatrani, 1998]. This relationship is shown in Figure 5.10.

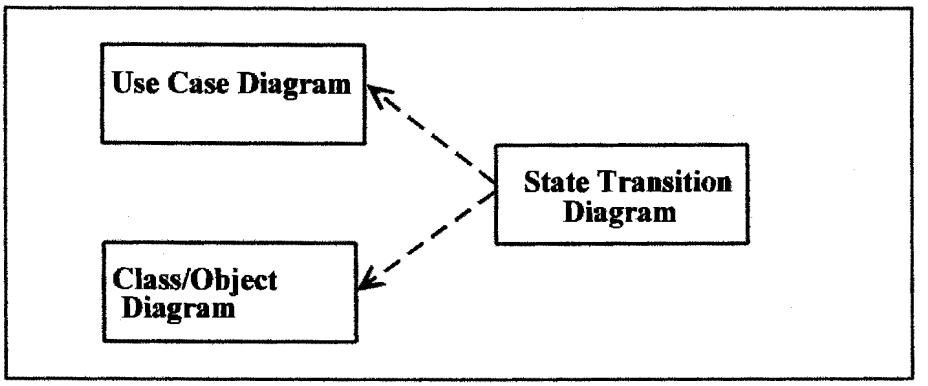


Figure 5.10 Use Case / Class Diagrams' Relationship to State Transition Diagram

Objects are associated with sequence and collaboration diagrams. Objects pass messages to other objects or themselves. Methods act as public interfaces to an object. In response to a message, in essence the calling of an object's method, an object performs some task and may or may not alter its state. An object may return a value that affects another object's state.

The internal behavior and control structure of a method is illustrated through activity diagrams. Activity diagrams illustrate the control structure of a method in much the same way as flow-charts illustrate the internal operation or control structure of a function in a procedural language. Figure 5.11 provides for the relationship of an activity diagram to a class diagram.

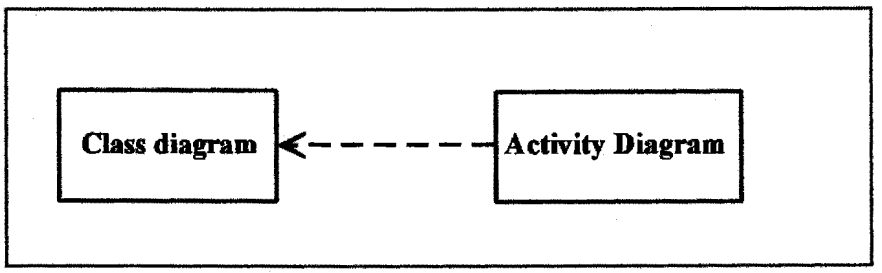


Figure 5.11 Activity Diagram's Relationship to Class Diagram

From the above diagrams one can see that there is a definitive relationship between different UML diagrams. To reenforce the concept of integration within UML, Figure 5.12 presents a model of all of the UML diagrams illustrating their relationship to one another.

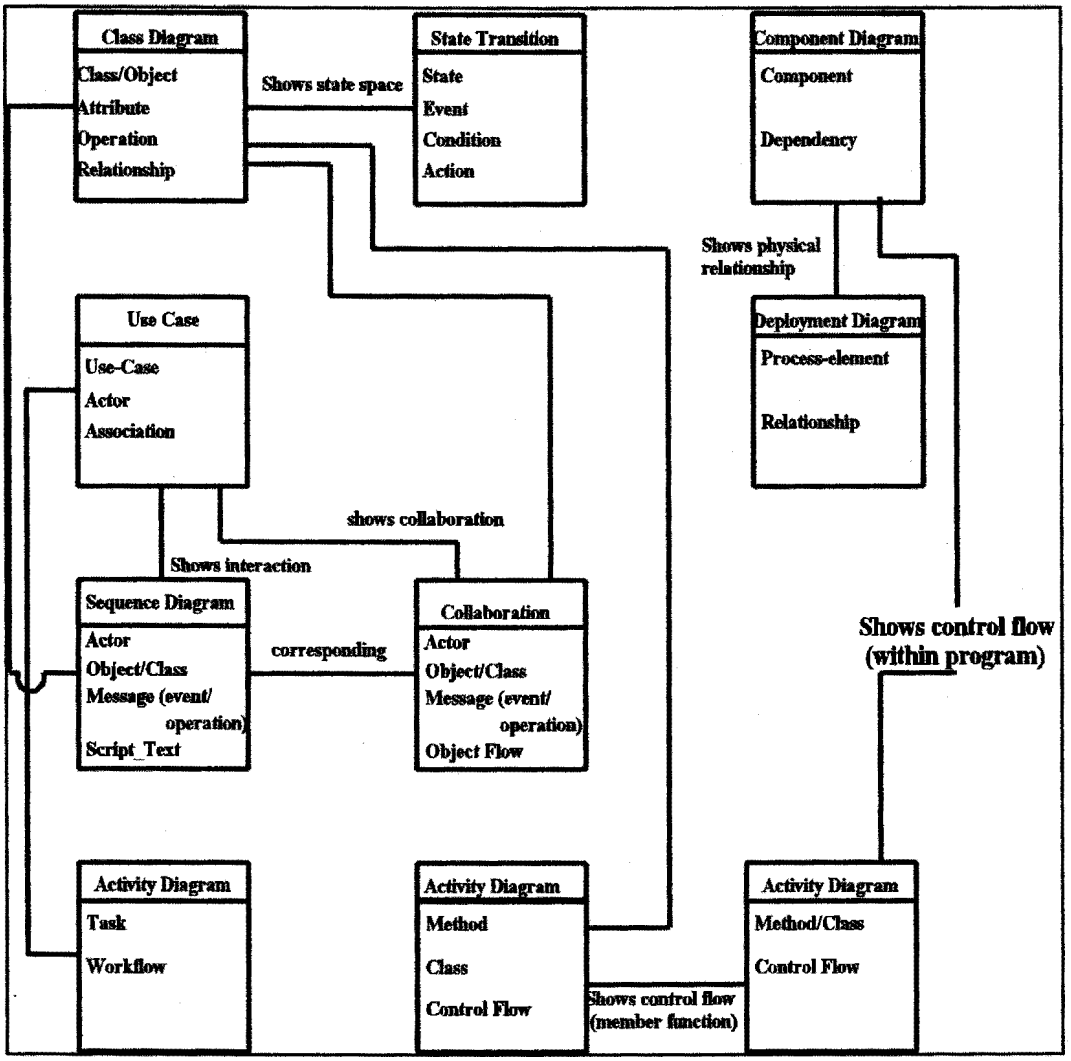


Figure 5.12 UML Diagram Interrelationships

Each of the diagrams represents a specific view of the system under development. As one can see, each view uses a different aspect of the class/object to model the software. Starting with the class as a unit, one can begin to add more detail and complexity. The relationship between classes

is presented with class diagrams where association, dependency, and specialization are illustrated. The dynamic aspects of the object interaction are presented with the use of collaboration and sequence diagrams. Detail of the internal operation of the methods is mapped out by the activity diagram. The component diagram is used to show the overall composition of the system based on software modules. By themselves none of these individual diagrams provides an adequate representation of the system under development. It is only when one examines the different diagrams and their interactions that you have a comprehensive view of the system. A combination of models is required to represent the software under analysis and design. To fully comprehend a class or object it is necessary that one examines that class as it is displayed in each of the many diagrams.

5.5 Conceptualization of Integration Testing Technique

Employing the Revised Spiral Model as the process model for developing object-oriented software reinforces the idea that object-oriented software follows an iterative path in its development. To properly develop object-oriented software one must have an adequate method for testing the software and for verifying that the design was implemented. This testing method should seamlessly merge with the iterative development process. For this reason the following integration testing technique was developed.

Building on the idea of the usefulness of applying post-implementation diagrams generated from the actual software for integration testing is examined. As defined in Section 5.2 integration testing of object-oriented software focuses on the intercommunication between objects and between components, whether at the object level or the sub-system level. Using post-implementation diagrams to identify the communication linkages can assist the tester in locating defects in the software, as well as, in developing test cases. Furthermore, the post-implementation diagrams can be compared with pre-implementation diagrams to further isolate defects in the

implementation and deviations from the design.

Essentially the integration testing technique is as follows: produce diagrams representing the structure and behavior of the system design and then compare them with equivalent diagrams derived from the actual software until a point of equilibrium is achieved. At that point use test cases derived from these diagrams and complete integration testing by exercising the software and examining the results, making changes to the design and system as required. The technique integrates with the Revised Spiral Model since the analysis of the software and the generation of the diagrams follow those points in the life cycles where individual object and sub-system integration occurs. The diagrams are developed whenever a new build is completed. The diagrams represent the iterative development of the software and as the software moves toward completion the post-implementation diagrams also become more complete. Changes in the software, brought on by defect correction and clarifications in the software, are reflected in the post-implementation diagrams. These diagrams are helpful in understanding the physical structure and behavior of the software, particularly in situations where the software design is lacking.

Integration testing of object-oriented software requires both the testing to code and testing to specification. The integration testing technique examined in this thesis is aimed at both of these requirements. The first half of the process is aimed at verification of the design specifications, whereas, the second half is concerned with the actual code. Comparison of the diagrams representing the design and the implementation views will reveal flaws.

The use of the technique provides an additional safety check where defects are highlighted. The earlier defects are found the less problematic they are to remove. Furthermore, identifying faults in the design can assist in prevention of potentially flawed systems from being released. Analysis and design defects can be quite expensive and the risk of their being allowed into the implementation must be taken seriously. The maturity of the software development process has a direct bearing on the probability of a flawed design being released into production. With the

use of this technique it is strongly suggested that there will be less likelihood that such an event can occur.

Integration testing, as mentioned in Section 5.4, occurs after objects are included in the new builds and after sub-programs are added to the system. To assist in proper testing at these points it is necessary that one have an understanding of the architecture of the system. The post-implementation design is helpful in developing test cases, however, depending upon the level of detail provided in the design diagrams, additional information may have to be gleaned from the software itself.

The design can be obtained from the software by applying reverse engineering techniques. This design, composed of UML diagrams, can then be compared with the original design documents. Variations in the diagrams are indicators that additional attention is needed to determine the cause of these variances. Certain variations are expected when considerable freedom is granted to the programmers. However, weaknesses in analysis and design can cause failures or inconsistencies within the implementation.

As discussed in Section 5.2, integration testing of object-oriented software entails developing knowledge about the interaction between objects, as well as components, within the programs. This information is obtained by analyzing the diagrams created from program executions.

At present, the source code is instrumented with statements which, once the software is recompiled, will generate data as the program executes. Additional information is obtained by tracing program execution by the use of debugger. The data gained by these two techniques can then be used to produce diagrams representing the logical and dynamic views of the software under consideration. Figure 5.13 provides a diagram of the steps taken to generate diagrams and related data from an object-oriented program.

One can compare the post-implementation diagrams with the pre-implementation or design

diagrams. Furthermore, these post-implementation diagrams can be used to assist in the development of test cases. Inter-object communication is provided in detail in the diagrams.

Defects can be isolated through the use of careful comparison of the pre- and post-implementation models. To identify the existence of a defect one takes the pre-implementation diagrams and visually examines the various artifacts to make certain that they are equivalent in the implementation diagrams.

If one uses a CASE tool to generate the class diagrams for a particular object-oriented system the initial software will be equivalent to the design. However, as the system evolves changes may be made to the code which alter class names, attributes, or operations. Any of these changes will be reflected in the post-implementation diagrams. Examination of the diagrams, as shown below, will point to changes in the class implementation.

For example, certain changes made to a class will have an impact on the software, such as, changing an attribute from a *long integer* to an *unsigned long integer* during implementation. This change may or may not impact on the operation of the software or on integration testing. However, this modification must be noted and further investigation carried out.

An important change would be the alteration of a class method where an additional parameter is included in the implementation. One's investigation must determine if the additional argument was indeed required, and what impact it has on already existing classes and objects. If the change is deemed necessary then the design must be updated so that further software development will reflect this modification. However, if the change is actually a result of misunderstanding or misimplementation of the design, then the software must be corrected so that it properly reflects the design.

The same method of comparison of pre- and post-implementation diagrams is necessary to

identify the presence of defects. Data are collected as the software executes. These data are then used to model the dynamic aspects of the software. Following the use cases developed during the analysis phase is helpful in creating test cases for exercising the software. These use cases are reflected in the design diagrams. Execution of the software and subsequent creation of the diagrams will provide the elements necessary for proper comparison of the diagrams.

Refinement and clarification will occur as the various test runs are completed. Data representing the various test scenarios are developed during the test planning phase. Additional insight into the architecture and behavior of the software will occur as different execution runs are completed. It is recognized that the first execution of the software, after it has been instrumented and recompiled, will not reveal all executable paths. In fact, since much object-oriented software is event driven, it is very difficult to initially develop extremely comprehensive test suites or scenarios.

Each iteration of program execution will produce additional information helpful in understanding the software and in recognizing defects in the software. State-diagrams, used to show object states, transitions, events, and actions, are developed which assist in establishing a pattern of data values included in messages to various objects. These data ranges are an important aspect of integration testing since it is common to locate flaws in software near the limits of acceptable values.

The dynamic behavior of object-oriented software is represented in iteration diagrams: state, sequence, collaboration, and activity. These diagrams can be produced by exercising the software. Even in the analysis and design phases these behavioral diagrams may not thoroughly document the desired action.

5.5.1 Comparison of Diagrams

In order to apply the technique it is important that one understands what is actually meant when it is stated that one should compare the pre- and post-implementation diagrams. Specifically, one is concerned with examining the static and dynamic aspects of an object-oriented system. The static view can be represented by class diagrams. Therefore, one should take the pre- and post-implementation class diagrams and compare each individual class (attributes and methods) and their associations with other classes. In particular, one is concerned with the methods and their specific interfaces and data types. These diagrams reveal a lot about how one object is able to communicate with other objects. If these pre- and post-implementation diagrams disagree then one should immediately look for the cause of the disagreement, before those classes represented in these diagrams impact negatively on the software being developed.

As for the diagrams used for modeling behavior, one must direct one's attention toward the sequence, collaboration, and activity diagrams. The use case diagrams are helpful in modeling the environment, however, it is not possible to precisely reconstruct the use case diagrams since external actors are not always represented in the source code. These use case diagrams, however, are useful in the development of integration and system test cases.

When examining the sequence diagrams one looks for differences in the order of object interaction and content of the messages. An incorrect message includes both the object and the message, along with the content's order and type.

The post-implementation diagrams will reflect the actual arguments along with their values while the pre-implementation diagrams only illustrate what arguments are expected. Therefore, one will have to convert the post-implementation diagrams into a format corresponding to the design diagrams. At present this conversion will have to be performed manually. One takes data obtained from the program during execution, along with analysis of the source code, and determines data types and names. From this information then the sequence and collaboration diagrams are developed. After the post-implementation diagrams are constructed then the

analysts and testers, along with assistance from the programmers, perform a diagram review and comparison with the diagrams produced by the CASE tools during the analysis and design phases. As the review proceeds the differences between the diagrams will be noted. If these differences are due to the process required to reverse engineer the diagrams then those differences will be ignored. Critical differences will be duly noted and the programmer, analyst, and designer will work together to determine the cause and location of the defect.

In certain instances the integration defect may be caused by a defect within a class's method. The detection of these types of defects falls under the topic of unit testing. However, if such a defect does occur and it is discovered through use of the integration testing technique, then it may be necessary to reverse engineer a class's method or methods.

Collaboration diagrams may also be reversed engineered through data collection during program execution. Collaboration diagrams are yet another means of describing program behavior.

Collecting data during program execution will provide more than ample data for program modeling. In fact, there will be instances where there is too much data being generated and one will have to reduce the volume in order to develop the corresponding behavioral data. Such is true when one encounters a loop which causes the creation/destruction of a list of objects or one is processing data related to a set of objects as in the use of an array or other type of container.

In regard to the sequence and collaboration diagrams the external actors will have to be included through observation since that information is not available from instrumenting and executing the program. Internal actors can be obtained by examining the data from the execution of the software.

The actual technique of comparing and identifying defects is based on looking at the overall structure of the diagram and highlighting all of the classes and objects which reside in the pre-

and post-implementation diagrams. Any object/class which does not exist in both diagrams will be easily identified. Immediately there will be a need for determining the cause for missing/extra classes and objects.

Once this introductory task has been performed then one examines the order of events in the software diagrams looking for invalid and incorrect messages and differences in the order in which objects sent messages. The comparison of the diagrams is currently performed manually. One examines the diagrams representing the same views of the software and highlights on both diagrams any place where there is disagreement. In most instances the post-implementation diagrams will provide detailed information regarding the location of the difference. In certain instances one will be able to determine the cause or need for the difference by examining the highlighted differences. Where the cause is an incomplete or incorrect design one will have to return to the analysis phase.

5.5.2 Steps for Integration Testing

It is logical that to properly test software one must have multiple views of the actual software. In the design each model represents a particular focus. The same is true for the models developed from the software itself, whether static or dynamic.

Taking the individual UML diagrams representing different views one must integrate them into a single unit. As the pieces are created they only give a limited, though important, view of the software. Once all of the diagrams are drawn then they can be reviews as a whole. Comparing the post-development picture with the one created from the analysis and design allows one to verify the design for proper implementation.

To clarify this concept, discussed in Section 5.5, it is necessary that there be a set of steps one follows in performing integration testing of object-oriented software. The following steps show

how one should carry out integration testing of object-oriented software.

The steps presented in Table 5.3 are shown in a flow diagram which provides the flow of analysis that starts with the source code and continues through the use of data generated from the system as it executes. Diagrams representing the implementation are generated throughout the life cycle until the system is fully integrated and is ready for system and acceptance testing.

Figure 5.13 illustrates the testing technique as a flow diagram which shows how the various steps are related and the sequence in which they occur. The flowchart, Figure 5.13, provides one with a sense of the iterative nature of the testing technique which corresponds with object-oriented software development under the Revised Spiral Life Cycle.

how one should carry out integration testing of object-oriented software.

The steps presented in Table 5.3 are shown in a flow diagram which provides the flow of analysis that starts with the source code and continues through the use of data generated from the system as it executes. Diagrams representing the implementation are generated throughout the life cycle until the system is fully integrated and is ready for system and acceptance testing.

Figure 5.13 illustrates the testing technique as a flow diagram which shows how the various steps are related and the sequence in which they occur. The flowchart, Figure 5.13, provides one with a sense of the iterative nature of the testing technique which corresponds with object-oriented software development under the Revised Spiral Life Cycle.

Table 5.3 Steps in Integration Testing Technique

1. Document the analysis and design of the software through the use of a CASE tool, creating diagrams representing all relevant views of the system. These diagrams will be used in Step 4 of the technique.

2.Develop preliminary test cases. These cases are developed from the information gained during the analysis and design phases and are used for the initial integration of the units

3. Reverse engineer the source code and produce detailed class diagrams. The class definitions in C++ will map to the class diagrams in UML. The use of inheritance in C++ will be used to complete the class diagrams.

4. Perform dynamic analysis of the executing software and produce state transition, collaboration, and sequence diagrams. As the individual member functions are called, specific data must be collected regarding the messages being passed to the member functions. When the member function has completed executing data are collected corresponding to the returned messages from that member function. In C++ member functions are called with specific arguments and results are returned from those member functions.

5.Compare the original pre-implementation diagrams with the corresponding post-implementation diagrams, highlighting any flaws and/or areas of deviation. The results of the comparison are used to correct any problems in either the design documents or in the software. One can also revise the class diagrams during this step.

6. Use the information from the implementation diagrams, after verification with the design models, to refine and expand the test cases developed in step 2.

7. Apply the integration strategy by exercising the software using the test suites from step 6. Again, as the software is run continuous collection of data will occur which will in turn be used in step 4. This iterative approach is consistent with the object-oriented model. Furthermore, it is essential to continually test during the development cycle to identify flaws introduced during coding.

8. Evaluate the results of the test executions. Defects in the software or design are identified and results are returned to the design and programming staff for correction.

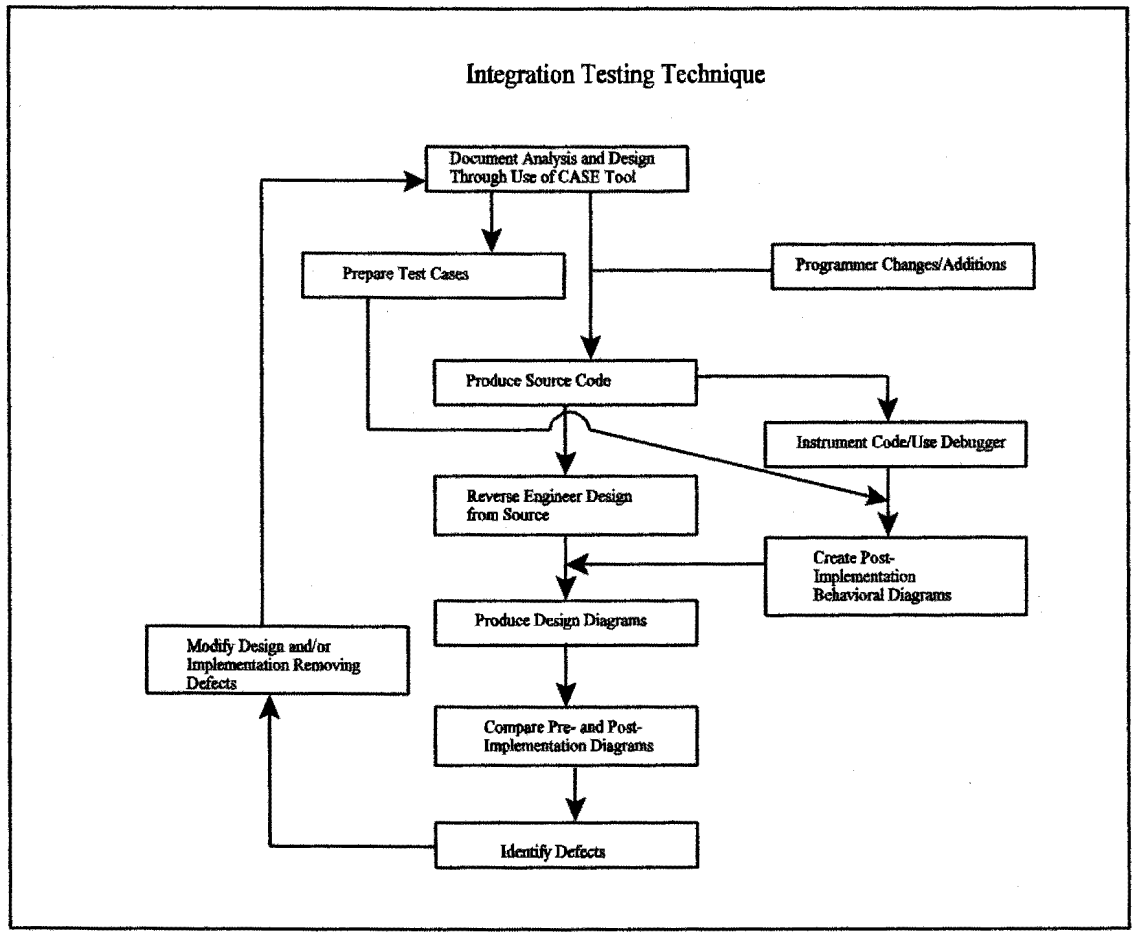


Figure 5.13 Flow Diagram of Integration Testing Technique

As the first units (classes, objects, and subprogams) are released to the build the integration testing process begins by building diagrams from the source code. Reverse engineering the class diagrams is provided through the use of CASE tools. As modifications are made those changes can be reflected in the diagrams. Preservation of the original diagrams is important since improper and incorrect changes to the design made during the implementation can possibly remain undetected if the original design is allowed to be arbitrarily updated. It is obviously important that the eventual design model be equivalent to the implementation model. Only after careful analysis and comparison of the pre- and post-implementation diagrams should the design diagrams be

modified. Arbitrary changes to the design places the testing group in a precarious position since the implementation becomes the design, and not visa versa. During the development life cycle controls must exist that prevent either a flawed design or flawed implementation from dominating the eventual system.

During the integration testing phase initial test cases are created that focus on testing to specification. As units and components are first released to the build, these preliminary cases are run and the first dynamic views of the software will be produced. Once there are results from the execution of the software, diagrams will be created that represent the implementation. The post-implementation diagrams can be created by a number of different means. At present, for purposes of this thesis, the code is instrumented to produce execution traces. In the best of cases a tool would exist capable of drawing the diagrams directly from data collected during execution. Modifications made to the compiler would provide for trace information generation in a manner similar to a debugger. After consistency between the design and implementation is achieved more detailed test cases are developed as the implementation proceeds. These cases are used for testing the implementation through the use of new and regression testing.

Examples of potential defects resulting from incorrect implementation, regardless of the original cause are listed in Table 5.4.

Table 5.4 Integration Defect Types

Type 1 - Improper Inheritance
Type 2 - Improper Message Sent
Type 3 - Incorrect Reply
Type 4 - Improper Object State
Type 5 - Incorrect Timing
Type 6 - Incorrect External Interface
Type 7 - Incorrect Sequence of Events
Type 8 - Incorrect Collaboration of Objects

Sequence diagrams can assist in locating faults where objects are not communicating in the right order. This type of defect is observed when one fails to declare destructors *virtual* when inheritance occurs. The base destructor will be called but not the destructor for the derived objects.

5.6 Software Development Scenarios Related to Integration Testing Technique

There are four different scenarios which have been identified which can lead to discrepancies between the design and the implementation. Unit testing is assumed to have been completed in each instance. Table 5.5 introduces four different scenarios where defects can be introduced into the system. A fifth scenario describes the situation where the design and the implementation are correct, complete and correspond to each other. This fifth scenario places a burden on the integration testing technique since it is impossible to demonstrate that a design and implementation are without defect. However, if the pre- and post-implementation diagrams correspond to each other then one can move to more traditional system testing in order to identify any defects within the system.

Table 5.5 Defect Introduction Scenarios

Scenario 1	Design is correct and detailed enough to begin implementation. Programming mistakes result in integration defects.
Scenario 2	The design is incorrect. Programming is done according to the design. Faults may also be introduced here as in Scenario 1. Integration faults will appear during program execution.
Scenario 3	Design is incorrect and programmers try to correct it. Integration defects may emerge. The implementation may contain defects resulting from both the design and the programmers' efforts.
Scenario 4	Design is correct but not detailed enough. Programmers address the detailed design and implement it. Here faults may be introduced again, as in Scenarios 1 and 3. Programmers may introduce defects, in this case, due to the lack of detail design information.

In certain instances one could argue that there is limited difference between scenarios 2 and 3 since the end result may be defective implementation. However, the cause of the defects has a marked impact on continued system development. If the source of defect introduction is the design itself then the effort to correct the implementation may be extensive if one has to continually modify the code as it is being written. It is logical that one seeks out the ultimate cause of the defects, eliminate that cause by correcting either the design or the implementation.

5.6.1 Scenario 1 - Design is Correct, Programmers Introduce Defects

Under this scenario the design diagrams are correct. In particular concern, however, is the fact that the programmers do not follow the design or else decide that the system needs enhancements or features beyond the design. Since the software does not correspond to the design then, when the post-implementation diagrams are derived from the software, the diagrams will not correspond

to the pre-implementation diagrams. It will be evident that there is a problem either in the software or in the design. Figure 5.14 provides a Use Case diagram illustrating Scenario 1.

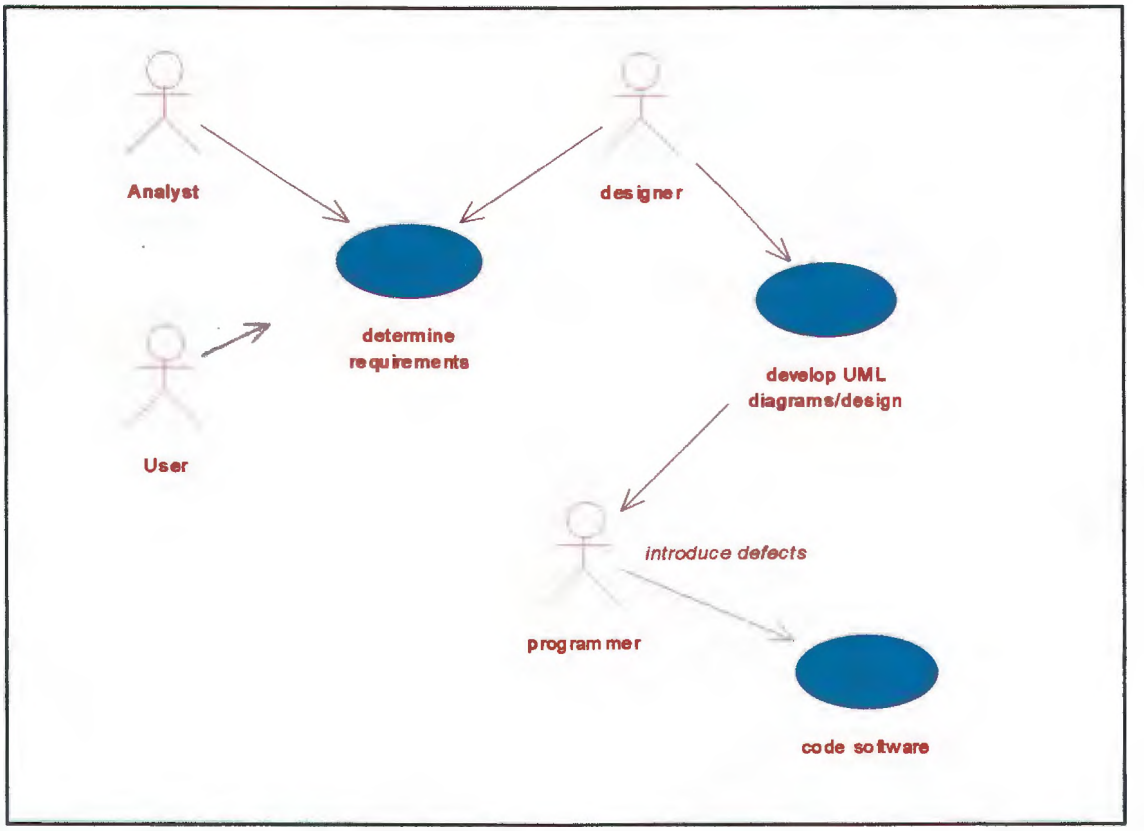


Figure 5.14 Design is Correct, Programmers Introduce Defects

One particular concern, following this scenario, is the modification of classes and member methods, particularly the method interfaces. As the software is developed the programmer has the freedom to modify the source code, even if the code was produced through the use of a CASE tool. The use of a CASE tool assists in software analysis and design, however, it does nothing to prevent the introduction of defects by the programmer once the code is released for further enhancement and for addition of modules deemed necessary after testing.

Furthermore, CASE tools, like Rational Rose, allow for the addition of notes for the purposes of documentation. These notes, particularly in connection with sequence and collaboration diagrams, may require additional code outside of the capability of the tool to generate. For this reason, it is important that one constantly monitors the progress of the software being developed and that one compares the implementation with the design through the use of diagram comparison and analysis.

As in Scenario 4, where the design is correct though incomplete, it is virtually impossible to know that the design is without flaws. For this reason the evaluation of the pre- and post-implementation diagrams must be thorough in all instances, whether or not one believes the design to be complete and without major defects.

Steps 1 through 5 of the technique would have highlighted inconsistences between the pre- and post-implementation model and would aid in identifying problems in the implementation. Repeating these steps, along with eliminating the faults which have caused the differences, would assist in removing such potential integration defects. Completing steps 6, 7, and 8, in Table 5.3, are necessary for integration testing.

5.6.2 Scenario 2 - Design is Incorrect, Programmers Follow Design

The design is incorrect and the programmers implement the software as designed. Figure 5.15 illustrates how this scenario may occur. In this instance the design and post-implementation diagrams are equivalent. However, the process of examining the diagrams provides the testing and development groups with the opportunity, which may or may not be successful, to identify the defects in the design and thus in the implementation.

Faults might also be unintentionally introduced here. Integration faults will appear during

program execution. Comparing pre- and post-implementation diagrams not only aids in locating integration defects but may also highlight the flaws in the original design.

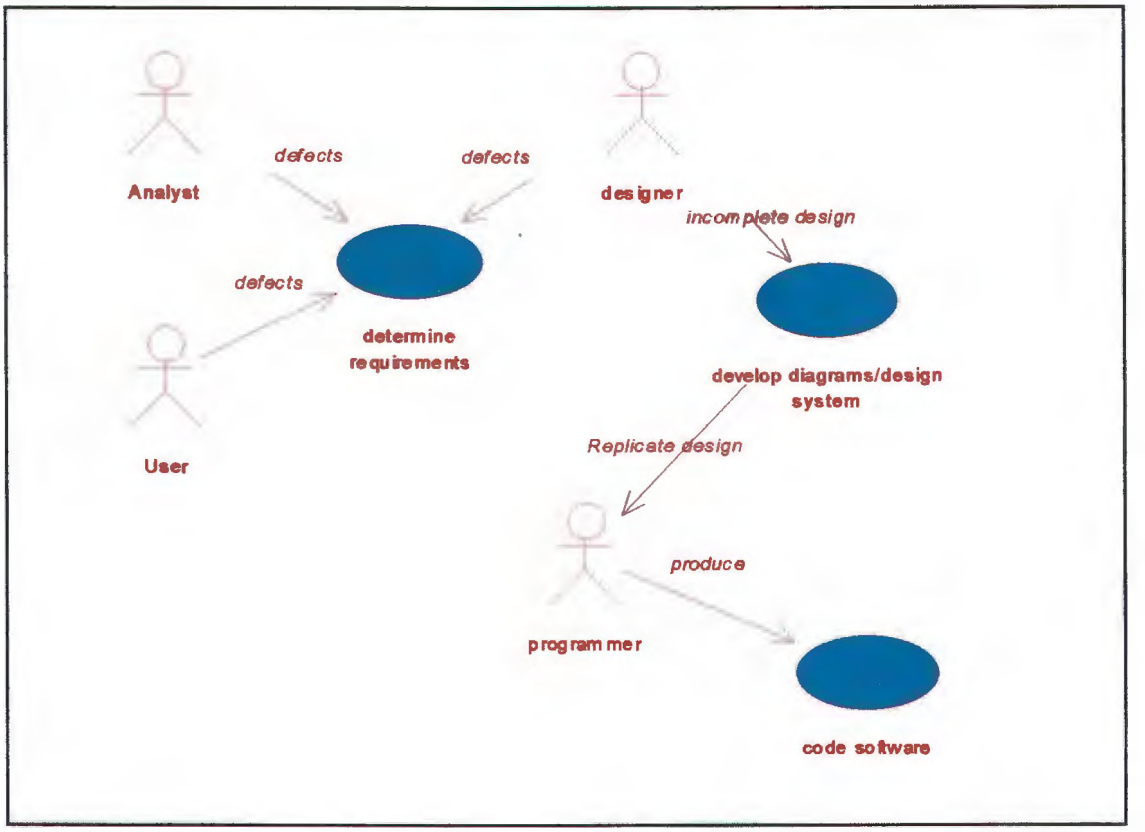


Figure 5.15 - Design is Incorrect/Incomplete - Programmers Implement Design

Modifications made in correcting the design and implementation will reduce the number of existing integration faults which were inadvertently introduced into the implementation as a result of an incorrect design. All of the types of integration defects identified in Table 5.5 may occur because of an incorrect design.

The comparison of the pre- and post-implementation models works as an extended code review since the testing group is actively reviewing the design and the implementation as provided in the

diagrams. Inspections and code reviews are an essential part of developing quality software [Perry, 1991].

One can argue that Scenario 2, as illustrated in Figure 5.15, may be divided into two separate scenarios as follows:

1. Design is incorrect and the implementation follows the design, therefore, the resulting pre- and post-implementation models show no differences.
2. Design is incorrect and the implementation does not model the design.

Both of these scenarios result in an incorrect implementation. In instance number 1, steps 1 through 5 would not reveal the design faults. Following steps 6 through 8, however, may highlight those instances where there are defects in the software which would possibly lead to changes in the implementation resulting in disparities between the design and the implementation models and requiring one to revisit steps 1 through 5.

5.6.3 Scenario 3 - Design is Incorrect, Programmers Attempt to Correct It.

The design is incorrect and programmers try to correct it. Integration faults may emerge. The technique reveals discrepancies between the design and the implementation. Further examination of the pre- and post-implementation diagrams may point to faults in both the design and the implementation. Since the design was incorrect, attempts to correct the design may result in either corrections or different defects being introduced, other that the one contained in the design. Simply recognizing the presence of defects (differences between the design and the implementation) will not resolve the issue of what needs changing. To solve this dilemma concerted effort should first be directed towards verifying the design. From there one can move towards comparing the diagrams representing the design with those generated from the implementation. Changes in the implementation should only be occur after one is satisfied with

the design. Modifications to both the design and the implementation may be required to reduce integration defects.

Once the programmers begin to develop the software, other than that generated by the CASE tool, it is time to begin reverse engineering the software and to begin the task of comparing the pre- and post-implementation diagrams. As soon as the two diagrams representing the two views are available then the process of inspection and review may reveal differences. Since the design is incomplete the programmer modifications may be apparent. However, as with the other scenarios, it is the responsibility of the analysts, designers, and programmers to determine where the defects exist. The differences are highlighted by the testing group and then those differences are turned over to those persons who have intimate knowledge of the design and the code in order to solve them.

As with Scenarios 2 and 3, all of the types of integration defects can occur as a result of a flawed design and an attempt by the programming staff to correct the design, as shown in Figure 5.16.

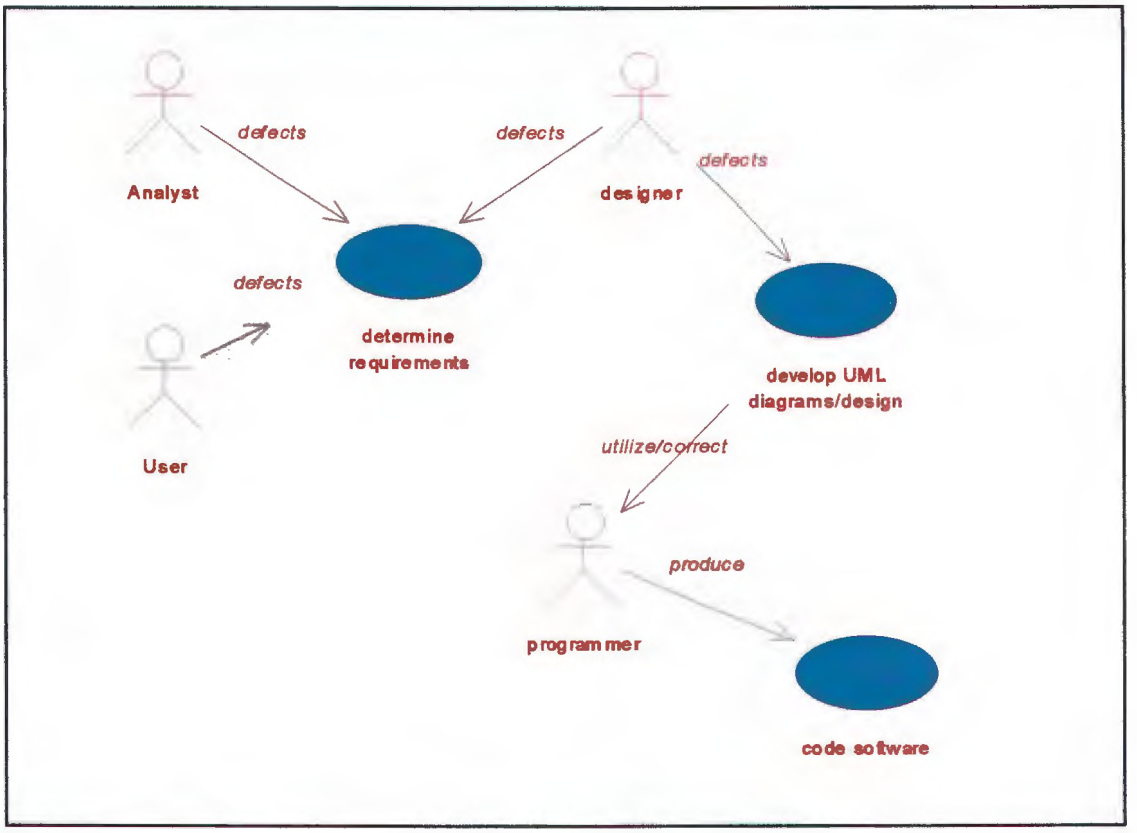


Figure 5.16 - Design is Incorrect, Programmers Implement, Attempting to Make Corrections

Steps 1 through 5 are used to identify inconsistencies between design and implementation the possibility of integration faults. Iterative correction of the differences, in this instance, would reduce the potential for integration faults.

Continuation of steps 6, 7 and 8 will aid in the identification of the remaining integration defects, as well as, possibly identify defects in the design and the resulting implementation.

5.6.4 Scenario 4 - Design is Correct, However Lacking Detail, Defects Occur

Design is correct but not detailed enough. Programmers address the detailed design and implement it. Here faults, as noted in Figure 5.17, may be introduced again due to lack of detailed design information. The results of this scenario are quite similar to those of Scenario 2 where programmers introduced defects into the implementation. The difference here, however, is that these faults are a result of a limited design resulting in programmers making incorrect decisions regarding the system's structure and behavior. Again, as in Scenario 2, all the listed types of faults may be introduced into the software.

The use of the technique, in this instance, will assist greatly in highlighting both weaknesses in the design and faults in the implementation. Deviations from the design will point to faults in the implementation. Additions to the design indicate areas where the design must be expanded. Changes in the design will eliminate some of the integration defects introduced by the programmers. Matching the implementation with the design will remove many of the commonly introduced integration faults and will place the pre- and post-implementation diagrams in the status of Scenario 1.

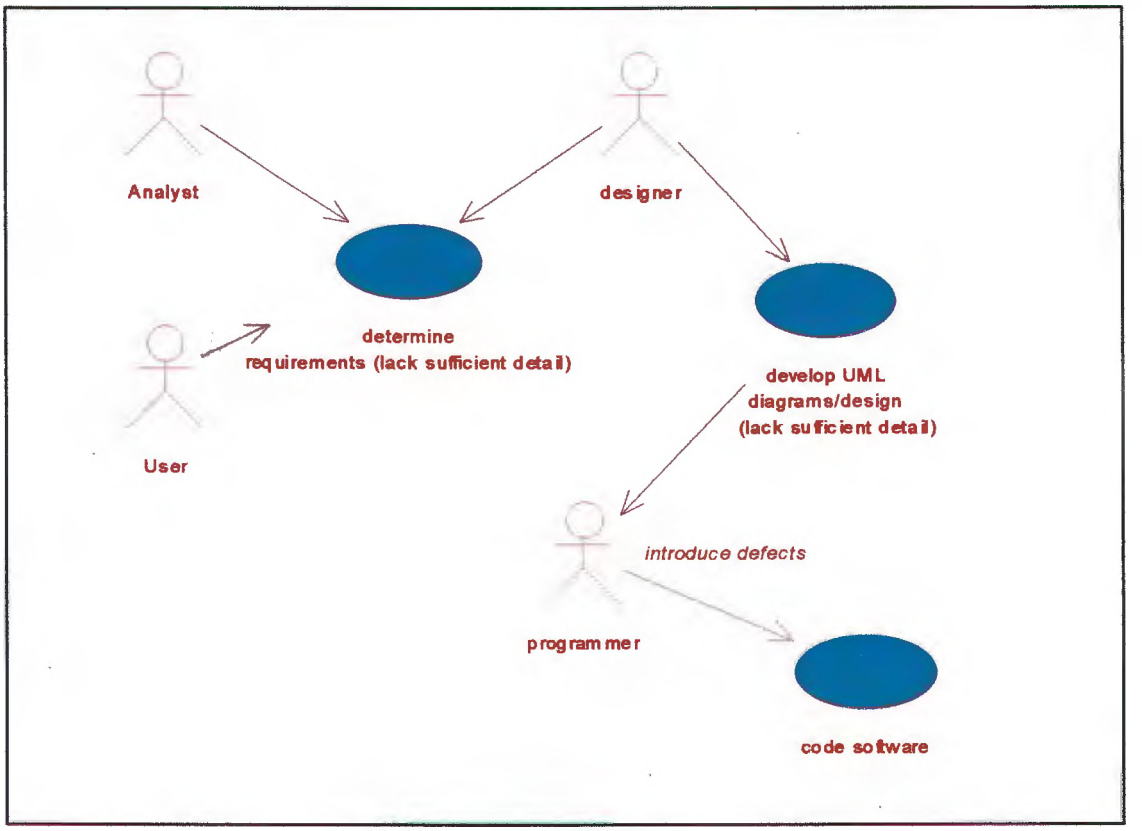


Figure 5.17 Design is Correct But Lacking Detail, Defects are Introduced by Programmers

Of particular interest in this scenario is the question of how the CASE tool handles the modeling of class methods and return values, especially within the context of sequence diagrams. If the CASE tool does provide for adequate documentation of return values, either directly or indirectly through the use of pointers and reference parameters, then non-code generating notes will have to be used. It is this type of incomplete design with prohibits the CASE tool from producing a complete software produce, leaving coding to the programmers who may misinterpret the design and allow defects to enter the software.

5.6.5 Scenario 5 - Design is Complete and Correct, Programmers Follow Design

The design is correct and detailed enough to begin implementation. Programming is rigorous and unit testing is completed. Units are then submitted to the builds. In this instance the pre- and the post-implementation diagrams would correspond. No significant integration defects are expected. The verification of the design is a plus increasing the level of confidence in the software. Steps 1 through 5 would be completed in this instance to verify that the pre- and post-implementation model are equivalent. Completion of steps 6, 7 and 8 are required to complete integration testing.

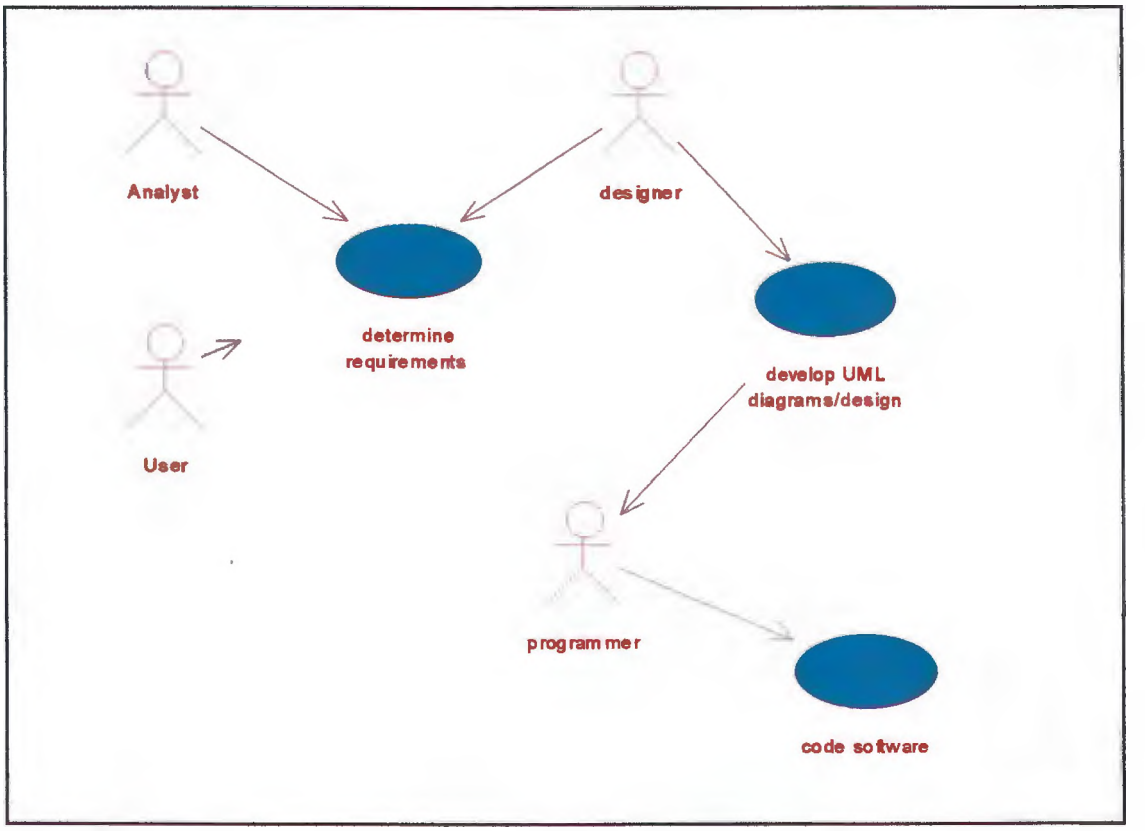


Figure 5.18 Design is Correct and Complete, Programmers Implement Design

Whenever the design is complete and correct and the programmers follow the design then one can

only determine from the technique that the design has been implemented, shown in Figure 5.18. In this instance one is in the same situation as when the design is flawed yet the programmers implement it without modification. Here one is then able to proceed to more traditional integration and system testing, utilizing the diagrams to create test cases.

Careful examination of the diagrams represents another form of code review and inspection. Instead of examining the code directly, examining the UML diagrams provides an additional view of implementation since the sequence and collaboration diagrams illustrate system behavior, something not available from just examining the static source code. Identifying defects represented in the diagrams provides one with a means of possibly identifying these defects prior to performing traditional integration testing. Furthermore, the iterative process of producing and comparing the diagrams will allow for review of the software whereas traditional code reviews and inspections are often limited in both their frequency due to the expense in time and manpower.

Scenarios 1 through 4 illustrate situations where the software is flawed, either because of an incorrect design and/or implementation. Weaknesses in the design can have a very negative impact on the ability of the programmers to code the software properly. If one simply relies on the initial design, necessary changes made to the implementation will not be reflected in the documentation and the development of adequate integration test cases will be impacted. Updating the design as the implementation progresses can also result in an incorrect implementation. Proper analysis and evaluation of the implementation, comparing it with the design, is essential if one is to identify defects. These defects must first be eliminated, and only then should the design diagram be updated.

To further clarify the scenarios, they can be expressed as a set of relations between the pre- and post-implementation models represented in corresponding models as portrayed in UML diagrams.

d = { correct, incorrect design models }

i = { correct, incorrect implementation models }

C = comparison of d and i where d and i represent the same diagram types

1. C(d,i) is true if d represents correct design and i represents correct implementation
2. C(d,i) is false if d represents incorrect design and i represents correct implementation
3. C(d,i) is false if d represents incorrect design and i represents incorrect implementation*
4. C(d,i) is false if d represents correct design and i represents incorrect implementation.
5. C(d.i) is false if d represents incomplete design and i represents incorrect or correct implementation*

*In terms of the above relations, an incomplete implementation is considered to be incorrect. Statistically it is possible for an incorrect or incomplete design and an incorrect or incomplete implementation to be equivalent. Only after integration and system testing of the software would this possibly become apparent.

To further clarify these functions, Table 5.6 provides a comparison among the various states of the design and the implementation and the possible state of the UML diagrams.

Table 5.6 Pre-Implementation/Post-Implementation Diagram Comparisons

	Design Complete	Design Correct	Implementation Complete	Implementation Correct	Diagram Comparison M=Match D=Do Not Match
1	✔	✔	✔	✔	M
2	✔		✔	✔	D
3	✔			✔	D
4	✔		✔		M/D
5	✔	✔			D
6	✔	✔	✔		D
7	✔	✔		✔	D
8	✔				D
9		✔	✔	✔	D
10		✔	✔		M/D
11		✔		✔	M/D
12		✔			D
13			✔	✔	D
14			✔		D
15				✔	D
16					M/D

Classification of terms is needed in order to understand what is meant by complete and correct when referring to the design and the implementation. A design is considered correct if there are no defects within the design that conflict with the system requirements or would prohibit the design to be functional if the design were implemented. However, when there is detail lacking

which may lead to incorrect decisions being made during implementation the design is considered to be incomplete. Additional lacking detail may lead to more than one implementation that follows the design. For example, the use of an array versus a linked list. The implementation may work correctly regardless of the type of container class used.

A design may be complete, covering all of the aspects of the system with adequate details which allow an operational system to be developed, however, since some of the system requirements are either left out or incorrectly stated, the design is not considered correct.

As one can see from Table 5.6 there are a number instances where the pre- and post-implementation diagrams will not match. Only with certain combinations of completeness and correctness will the diagrams possibly be equivalent. In Table 5.6 check marks are used to indicate when a particular condition is true, e.g. the design is complete, the design is correct.

Each of the combinations is explained as follows:

Row 1 - If the design is complete and the design is also correct then if the implementation follows the design and is fully implemented the diagrams representing the pre- and post-implementation will match.

Row 2 - If the design is complete, though incorrect, it is possible for modifications to be make during implementation which will result in a correct and complete system thus leaving the pre- and post-implementation diagrams not matching.

Row 3 - If the design is complete, though incorrect, it is possible for the implementation to be a complete system without being correct. Here the diagrams will not match.

Row 4 - If the design is complete, though not correct, and the resulting implementation is complete, and again not correct, the corresponding pre- and post-implementation diagrams may or may not match.

Row 5 - If the design is complete and correct and flaws are introduced during implementation resulting in a defective implementation which results in both an incomplete and incorrect system then the pre- and post-implementation diagrams will not match.

Row 6 - If the design is both complete and correct, and the implementation results in a complete system, though flawed, the diagrams will not match.

Row 7 - If the design is both complete and correct, and the implementation is correct, though elements are not implemented, then the pre- and post-implementation diagrams will not match.

Row 8 - If the design is not correct, though there is a complete system design, and modifications are made during implementation in a effort to correct these defects, the pre- and post-implementation diagrams will not match.

Row 9 - If the design is not complete, however it is correct, and the implementation is complete and correct then the pre- and post-implementation may be consistent, however, the additional elements in the complete implementation will result in the diagrams not matching.

Row 10 - If the design is correct, though there are no existing logical defects, the lack of completeness will allow for the introduction of defects which may result in a complete system which is incorrect. Both the lack of completeness in the design and the introduction of defects during implementation will result in the pre- and post-implementation diagrams

not matching.

Row 11 - If the design is not complete, i.e. certain detail is lacking, the implementation may result in a system that matches the implementation and it also incomplete. Here the pre- and post-implementation diagrams may very well match. Still, due to incompleteness of the design, alterations in the implementation may result in non-matching diagrams.

Row 12 - If the design is not complete, lacking detail, though correct and the resulting implementation is both incomplete and incorrect, the pre- and post-implementation diagrams will not match.

Row 13 - If the design is incomplete and incorrect and changes occur during implementation that correct defects in the design, though do not finalize the implementation, the diagrams will not match.

Row 14 - If the design is incomplete and incorrect and corrections are made during the implementation then the implementation may be complete though incorrect. In this instance the diagrams will not match.

Row 15 - If the design is incomplete and incorrect and corrections are made during implementation then the implementation may be correct, though lacking certain elements. This situation results in pre- and post-implementation diagrams not matching.

Row 16 - If the design is both incomplete and incorrect and the implementation follows the design the result would allow the pre- and post-implementation diagrams to match. However, any deviations from the design would result in diagrams that do not match.

Continually creating enhanced views of the implementation, updating the design, if necessary, and expanding the completeness of the tests being performed, will provide further assurance that the design is being reflected in the implementation. This evaluation, testing, and validation process follows the same iterative concept as does the overall development life cycle suggested by the Revised Spiral Model, and illustrated by Figures 5.4 and 5.5, thus adding to the overall integration of the life cycle.

Even though there are different scenarios which may lead to the introduction of defects into the software, the resulting defects may not be indistinguishable, that is, defects in the design do not necessarily produce a particular type of defect. However, in recognizing the cause of the defect, e.g. design is flawed or implementation is flawed, one may be able to introduce changes into the development process which will in turn reduce the number of future defects. For example, if one determines that the design is either defective or is lacking in detail (Scenario 2 and Scenario 4) then the proper reaction would be to place the programming activities on hold until either further analysis and/or design can be completed and the design defects removed.

5.7 Applying UML to Integration Testing Technique

As discussed in Section 5.5, the integration testing technique relies upon the comparison of diagrams representing the design and the implementation of an object-oriented program. The Unified Modeling Language (UML) is recognized as a widely accepted modeling language for object-oriented systems. UML, discussed in Section 5.4, contains various diagrams needed to represent the logical and physical models of object-oriented software. These diagrams support all of the views required for software design and for integration testing.

Furthermore, UML is moving through the process of becoming standardized, thus adding to its acceptance within the software development community. For this reason it is suggested that

UML be the language of choice for implementing the integration testing technique.

Presently, there exists a number of available CASE tools which support UML and the generation of C++ code, as well as, other commonly used object-oriented languages. These tools allow one to design a system and produce initial code. With the addition of reverse engineering functionality one is able to generate certain key models from the actual software.

Rational Rose, developed by the Rational Corporation, whose principal scientists are Rumbaugh, Booch, and Jacobson, provides such capabilities. For this reason, as well as, its direct link to the authors of UML, Rational Rose was chosen as the CASE tool for use in implementing the integration testing technique.

Since Rational Rose was selected as the CASE tool for assisting in the design of the prototype, it is important to briefly discuss how the tool works and its limitations. In particular is the concern for the generation of source code from the actual UML diagrams, and related documentation, and the reverse engineering of the design from the software.

Rational Rose 98i provides a separate tool for code generation. In this thesis C++ code generation was selected. The design documentation, UML diagrams, is used by the code generator to produce elements of the final system source code. The code generator, however, lacks the ability to produce detailed subroutines but instead produces structures for the classes. These structures properly identify the class (name) and its attributes and operations. However, the internal structure of the operation, i.e., decision paths inside of the operation, is left to the programmer. Granting the programmer such important work leaves open a critical path where defects can be easily introduced. Because of this weakness, the integration testing technique of comparing pre- and post-implementation diagrams is needed in order to identify and eliminate key system defects. Traditional integration testing will help find flaws. However, it will not always find major design defects since the process is focused on inter-method communication not

functional design issues.

The integration testing technique relies strongly on the ability to extract the design from the system. In order to do so, one must have a set of tools capable of understanding both the structure and the behavior of the system. Rational Rose provides an extension which is capable of analyzing, in this case, C++ source code. In doing so, the tool can produce a set of class diagrams representing the logical design. However, the tool is restricted to static analysis of the source code and does not support the creation of sequence, collaboration, activity, or state diagrams which are necessary to model system behavior. The creation of such behavioral diagrams, therefore, must be created through the use of some other tool or code instrumentation.

The following table illustrates the type of integration faults that can be detected by generating one or more of the specific UML diagrams.

Table 5.7 UML Diagram Correlation to Defect Detection

Defect Type	Diagram Type					
	Use Case Diagram	Class Diagram	Object Diagram	State Diagram	Collaboration Diagram	Sequence Diagram
1. Improper Inheritance		X	X			
2. Improper Message Sent					X	X
3. Incorrect Reply				X	X	X
4. Improper Object State				X		
5 Incorrect Timing						X
6. Incorrect External Interface	X				X	
7. Incorrect Sequence of Events						X
8. Incorrect Collaboration of Objects					X	

As can be seen in Table 5.7 the UML diagrams are useful in the identification of defects in the software in both the logical and physical models. Class and object diagrams help identify faults in the attributes and the methods belonging to a class or object. The state transition, collaboration, and sequence diagrams provide insight into how the various object communicate with each other in completing tasks. Many of the diagrams provide insight into similar defects. For example, detailed class diagrams can reveal a situation where a particular object is not returning the proper message due to the method processing the wrong data type. This same defect can be revealed through examination of the sequence and collaboration diagrams.

Table 5.8 provides an overview of the relationship of the testing technique and defect introduction scenarios discussed in Section 5.6. Each of the scenarios can introduce defects into the final product. Utilizing the technique provides one with a means of isolating differences between the design and the implementation. Defects can be introduced throughout the life cycle and it is important that adequate inspection and evaluation occur in order to identify the existence of defects and to remove those defects as soon as possible once their presence becomes known.

Of the four defect introducing scenarios, outlined in this thesis, Scenario 2 produces both a design and an implementation that is defect riddled from the design. However, only after examination and testing will their presence be made known since it is possible that the design will be logically sound while still not representing the user's real requirements.

Table 5.8 Integration Testing Technique - Defect Introduction and Detection

Use Case - Analysts determine the system requirements. Designers use these system requirements to design the system using UML diagrams to represent the system's structure and behavior. The programmers implement the system based on the design and the system requirements. UML diagrams are developed from the implementation and compared with the design diagrams in an effort to identify presence of defect

Precondition - No Defects	Postcondition - Defects Found
Scenario 1 The analysts and designers correctly identify system requirements and design a corresponding system. The programmers, however, either misinterpret, modify or incorrectly code the system.	Response - Numerous Defects Incorrect Class Implementation Incorrect Inheritance Incorrect Object Collaboration Incorrect Event Sequence Incorrect Object Messaging Incorrect External Interfaces
Precondition - Defects	Postcondition - Defects Exist But May Not Be Found
Scenario 2 The analysts and designers incorrectly identify system requirements and/or create a defective design. The programmers implement the design, potentially adding defects.	Response - Numerous Defects Incorrect Class Implementation Incorrect Inheritance Incorrect Object Collaboration Incorrect Event Sequency Incorrect Object Messaging Incorrect External Interfaces
Precondition - Defects	Postcondition - Defects Found
Scenario 3 The analysts and designers incorrectly identify system requirements and/or create a defective design. The programmers attempt to correct the design.	Response - Defects Incorrect Inheritance Incorrect Object Collaboration Incorrect Event Sequence Incorrect Object Messaging Incorrect External Interfaces

Precondition - No Defect	Post Condition - Defects Found
Scenario 4 The analysts and designers correctly identify system requirements and design a corresponding system. Detail is lacking from the design allowing for programmer introduced defects.	Response - Defects Incorrect Class Implementation Incorrect Inheritance Incorrect Object Collaboration Incorrect Event Sequence Incorrect Object Messaging Incorrect External Interfaces

[Table 5.8 Cont'd]

5.7.1 Integration Testing Technique - Relationship With Unified Software Development Process

The technique proposed in this thesis is intended to aid the tester in understanding the structure and behavior of the object-oriented software being developed. The technique utilizes structure and behavior to determine how and when components within the software interact. Components in this case can be either individual objects, member operations, or sub-programs in the sense that components are defined by the UML.

In order to integrate the testing technique with the software development life cycle it is important that one has a development architecture within which one can attach the framework for integration testing. For this reason the Unified Software Development Process is being used as a structure by which the integration testing technique can be utilized.

[Jacobson, et al., 1999] integrated UML into their Unified Software Development Process. Of specific interest here is their focus on testing during the life cycle. Four major artifacts of testing are identified:

1) test component,

2) test plan,

3) defects, and

4) test evaluation.

[Jacobson, et al., 1999]

Each of these artifacts are important in developing a test strategy for object-oriented software.

5.7.1.1 Proposed Test Model

As the software is developed, various units are tested and then, after acceptance, are released to the next build. It is at this point that integration testing occurs. The nature of the complexity of object-oriented software can cause the new element to have multiple relationships with other objects and components. Test cases must be built that test all of these relationships. Before the test cases can be finalized one must have a thorough understanding of the essence of the software. Only by analyzing the actual code and its execution can one properly understand how the software is constructed and how it behaves under varying conditions. The Test Model, proposed here, includes the step of comparing the design model with the implementation model. This comparison may point to defects in the design, the implementation, or both, and assists in the reduction of faults left for identification during integration testing.

5.7.1.2 Test Case

[Jacobson, et al., 1999] define a test case as a means of specifying what should be tested, when, and with what inputs and what results. Consideration is given to both the time, effort, and cost required to perform that type of testing. A major consideration, when evaluating a testing technique, is its effectiveness in identifying the presence of faults in the software. A test case with a low probability of finding defects is not justified unless those faults have a high probability

of being ignored by other defect detection techniques. In the proposed technique attention is drawn to identifying potential defects in the implementation and removing them before integration testing occurs thus, hopefully, increasing the effectiveness of integration testing. Test cases are developed from the accepted implementation model, which shows which components interact, and through the evaluation of the use case diagrams and related scenarios. Specific test cases are then used for more traditional integration testing at the inter-component level and eventually at the system level, after the software has been fully integrated and tested. The technique increases the level of confidence one has in integration testing since a number of the defects commonly found by integration testing are removed prior to actual testing occurring. The technique, founded on the concepts embodied in inspections and reviews, expands ones knowledge of the architecture and behavior of the software and aids in the identification of defects in the software and/or design.

5.7.1.3 Test Procedure

The test procedure for the proposed integration testing technique, is discussed in Section 5.5 and further outlined in Table 5.3. Essentially one creates pre- and post-implementation UML diagrams, compares those diagrams, highlighting any disagreements, investigates the cause(s), makes any necessary changes to the design/implementation, and repeats the process until agreement is reached. Once consensus is achieved then more traditional integration testing occurs using test cases developed from the system's use cases and information obtained from the post-implementation diagrams which illustrates component interactions.

5.7.1.4 Test Component

Within the Unified Software Development Process the test component is viewed as a means by which software testing is automated. The test component automates one or more of the test procedures. Because of the experimental aspect of the work in this thesis, the test component is

directed not at the automation of the test procedures but at the use of existing tools for the development of the UML diagrams and for the development of test cases.

Under the integration testing technique reverse engineering capabilities of a CASE tool are used to generate the static implementation diagrams representing the class diagrams. The creation of the dynamic models is more difficult.

At present, the UML diagrams representing the dynamic view of the software can be created in a number of different ways ranging from the simplistic addition of data production statements, *cout* in C++, to the creation of a tool capable of extracting the data while the program is executing and then automatically generating the UML diagrams. The final production tools of the test component, needed to create UML diagrams representing the dynamic view of the software, are beyond the scope of this thesis. In order for this integration testing technique to be fully operational in a production environment it is recognized that such an automated tool is required. For the purposes of illustrating the efficacy of the technique, a combination of data production statements and debugger tracing is employed. Future efforts will be directed toward the creation of a set of tools necessary to automate the entire process of code analysis, model production, and pre- and post-implementation diagram comparison.

5.7.1.5 Test Plan

Under the proposed technique the target is integration testing of components. The tests to be performed are based on the calling of the components in the various sequences (order) in which they have been identified by analysis of the software. Comparison of the implementation models with the design models provides the testers with a deeper understanding of how the software is constructed. Changes and defects in the software, made during implementation, may be identified which assists in the development of tests which provide for adequate system coverage.

Each time a build occurs, new diagrams should be created which reflect the latest software construction. As the software first begins to be integrated, the diagrams representing the implementation will be limited in scope. However, as the software progresses toward completion, more and more detailed diagrams will be produced which will assist in the creation of test cases and the execution of appropriate inter-component tests.

At present, the UML models reflecting the dynamic aspect of the software as it executes can be created in a number of ways. Through the use of a debugger one can apply breakpoints and trace through the software as it executes. Taking the trace information from the debugger one can then develop the diagrams either manually or by applying that information to Rational Rose. Another means of acquiring the information from the program as it executes, is to develop a series of macros that report information about object state each time communication occurs between components. Building a class that reports information about an object's state can also be developed which reports each time the object state is either reported or modified. At present, there does not exist a means of creating UML diagrams directly from the object code. Further research is needed before such a tool is made available. However, the lack of an automated tool does not invalidate the basic concept upon which the technique is built, that being that inconsistencies between the design view and the implementation view immediately signal the need for further investigation of the cause.

5.7.1.6 Defect

"A defect is a system anomaly." [Jacobson, et al., 1999] Whenever the system does not comply with the expected behavior or does not follow the system's design it should be recorded as a defect. Certain defects may prove to be incorrectly identified since they are results of a flawed design.

Under the proposed technique all deviations from the original design will be recorded as defects.

As further analysis of the software is completed, only certain defects will be reported as needing correction since clarification of the deviations of the implementation from the design may be explained through software inspection and review. Certain differences between the diagrams may be a result of increased data derived from analysis of the software as it is executing, thus expanding the amount of detail in the UML diagrams representing the implementation.

5.7.1.7 Test Evaluation

Test evaluation covers all of the analyses of results of the testing effort, from test-coverage, to code coverage, to the status of defects. This technique of developing UML diagrams throughout the implementation phase greatly aids in the evaluation of integration tests, code reviews, and system tests. As the code is evaluated, visual models of the software assist in understanding the structure and behavior of the software at levels which are used in integration testing of the components. The comparison of pre- and post-implementation models aids in the verification of the design. In addition, the process is much like a code review in that the diagrams represent the two models of the software. Careful comparison of the diagrams may provide insight into flaws in the design and/or implementation.

5.8 Prototype for Technique Evaluation

To evaluate the technique as described in Sections 5.5 and 5.6, a prototype of a project management system is developed. The purpose of the prototype is to provide a software system by which one can experiment with the integration testing technique, making changes to the prototype and seeding it with defects. Following the steps outlined in Section 5.5.2 it is possible to evaluate the effectiveness of the technique, that is, to evaluate whether the technique is able to help identify known defects in the system. The prototype is presented in Chapter 6 with the evaluation of this technique following in Chapter 7.

As expressed in Section 1.4.1 the prototype is limited to an object-oriented system written in C++. The prototype utilizes aspects of C++ that demonstrate object-oriented concepts: encapsulation, polymorphism, and inheritance. In addition, language elements unique to C++ are also included, e.g., templates. Together these elements provide for a more realistic test of the integration testing technique.

The prototype uses UML diagrams for the analysis and design of the system. A condensed version of the Revised Spiral Model is employed, focusing on the analysis, design, and implementation cycles. The steps presented in Table 5.3 and Figure 5.13 are followed as the prototype is developed. UML diagrams are created representing the logical and physical views of the project management prototype. These diagrams are used in the comparison of the pre-implementation and post-implementation models of the system.

5.9 Summary and Conclusion

This chapter provides the fundamentals of the proposed integration testing technique. UML is used to model the object-oriented software. The relationship of UML to the software development process is first discussed. Emphasis is placed on the need for using all of the different design models in order to properly analyze and design software under the object-oriented paradigm. Attention is drawn to the use of the same object-oriented models to facilitate integration testing of the software. Creating the various diagrams from the source and executable versions of the software is discussed as it relates to the integration testing of the software. The steps followed in implementing the technique are presented along with a flow diagram modeling the procedure. Additional testing must be carried out to determine precisely where defects are located. The types of faults often encountered in integration testing are classified in relationship to the type of defects each type of UML diagram can help to identify.

Chapter 6 uses the UML notation to develop post-development diagrams of the software. A

prototype is developed using UML for the design models and then compares the post-implementation diagrams with these models. Specific types of defects are seeded into the software to determine whether the technique is successful or not in identifying the existence of these faults.

CHAPTER 6

Demonstration of Concept

CONTENTS

6.1 Introduction

Chapter 5 presented the idea of comparing models developed from the actual software with corresponding models used in the analysis and design of that software for the purpose of integration testing of object-oriented software. In addition to comparing the design with the implementation, it was also suggested that the individual models would be helpful in building test

cases for performing integration testing. Repeatedly producing diagrams representing different views of the software helps one to further understand the architecture of the software. Object-oriented software, particularly such software developed in C++, relies heavily on the use of polymorphism as implemented through the use of generic classes, overloaded functions, and dynamic binding. For this reason, it is not sufficient to simply use the source code for building test cases. By performing dynamic analysis and reverse engineering of the executable code, it may be possible to improve on one's understanding of the software. Expanding one's understanding of the structure and behavior of software is especially crucial when an independent testing group is responsible for testing that software.

This chapter contains a demonstration of the proposed testing technique as applied to a small software system. The results are presented in this chapter. The following section describes the nature and construction of the prototype system.

6.2 Overview of Prototype System

The prototype system is a project management subsystem for a small software development group. There are four actors interfacing with the subsystem, namely the executive, the personnel manager, the project managers and the employees. The subsystem is intended to support each of these actors in their respective responsibilities:

1. The executive serves in an oversight position.
2. The personnel manager manages employee data.
3. The project manager manages the project data and assigns employees to projects.
4. The employee provides personal information necessary for employment.

The system functions to be provided by the subsystem are enumerated in the Function Table given in Table 6.1.

Table 6.1 Function Table for Prototype

System Function	Actor Involved	System Attributes	Constraints
Executive Oversight	Executive	Provide query of personnel data and software project data	Cannot alter system
Personnel Management	Personnel Manager	Add, modify employee data	Interacts with employee and personnel management data, cannot modify project data
Project Management	Project Manager	Create, modify software project data	Assigns employees to projects, cannot alter personnel data
Employee Interaction	Employee	Accept personal data for employment	Cannot interact directly with project management system

The project management subsystem is divided into four major divisions which correspond to the tasks performed by each of the actors. Depending upon the actor, there is limited or no access to the actual software system. As can be seen in Table 6.1, certain constraints are applied to each of the system functions.

The use case context diagram for the project management subsystem is given in Figure 6.1.

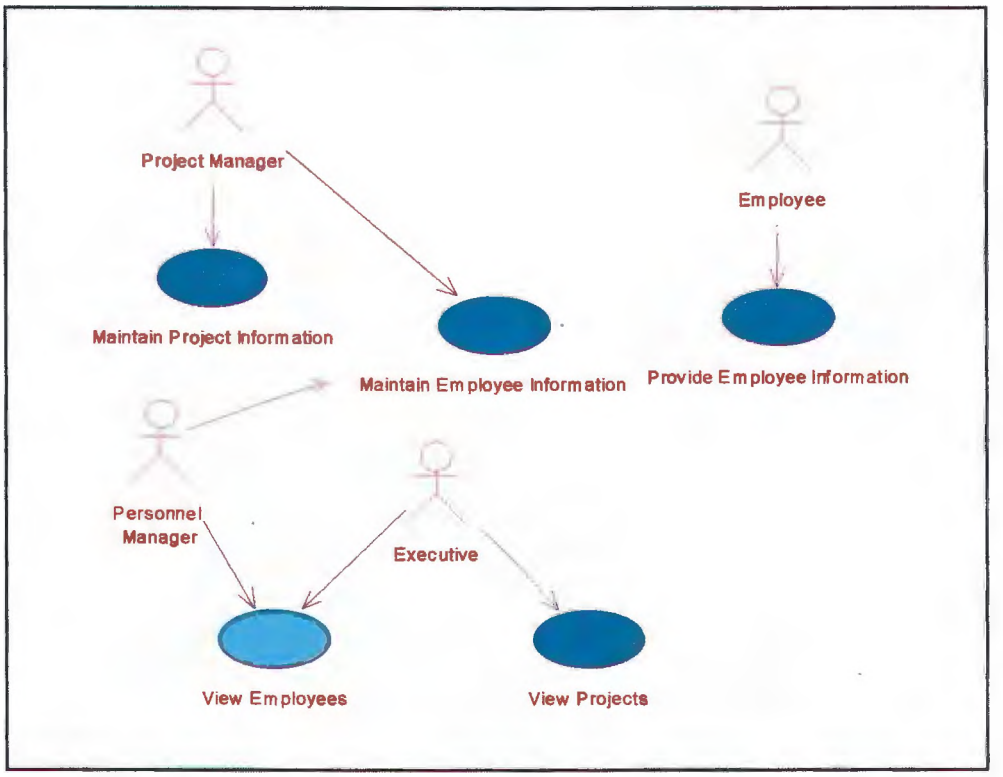


Figure 6.1 Use Case Diagram for Project Management Prototype

From this main use case diagram, Figure 6.1, one can see that the four actors are the employee, the personnel manager, the project manager, and the executive. The project manager is responsible for maintaining project information, similar in nature to the personnel manager who maintains employee data. When one speaks of maintaining data, the task includes additions, modifications, and deletions. The project manager interacts with the employee data when assigning an employee to a particular project. The executive, in the prototype, oversees both personnel and projects by examining their status, but is restricted from making any modifications to either. The employee, within the context of the prototype, provides personal identification

information to the personnel manager. The employee does not directly interact with the project management system.

Once the use case diagram has been developed, each of the individual scenarios must be modeled to provide necessary detail about the interaction of objects in the support of each scenario. As discussed in Section 5.4, there are two primary UML diagrams used to model system behavior - the sequence diagram and the collaboration diagram. Sequence diagrams provide for the major actions which can be performed by the various actors. Sequence diagrams are presented here for the *create project* and *create employee* supporting the scenarios represented by the use case diagram. In a complete system design, sequence diagrams would be developed for each of the scenarios. State diagrams would also be developed which would provide for event/data tracing, however, due to the limited nature of the prototype they are not required.

Use cases play an important role since they help identify the interaction of objects which were eventually translated into sequence diagrams. To help further illuminate the project under development, collaboration diagrams, Figures 6.2 and 6.3, were created that represent a slightly different view of how the various objects interact. Sequence diagrams, Figures 6.4 and 6.5, provide an additional means of viewing the design of the system.

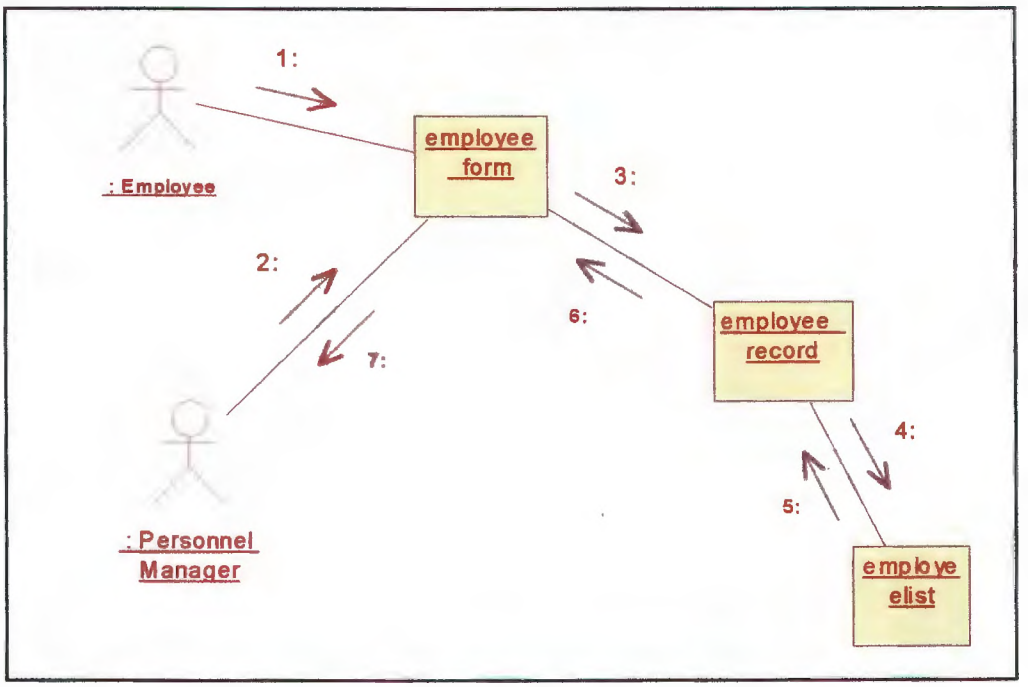


Figure 6.2 Collaboration Diagram for Create New Employee

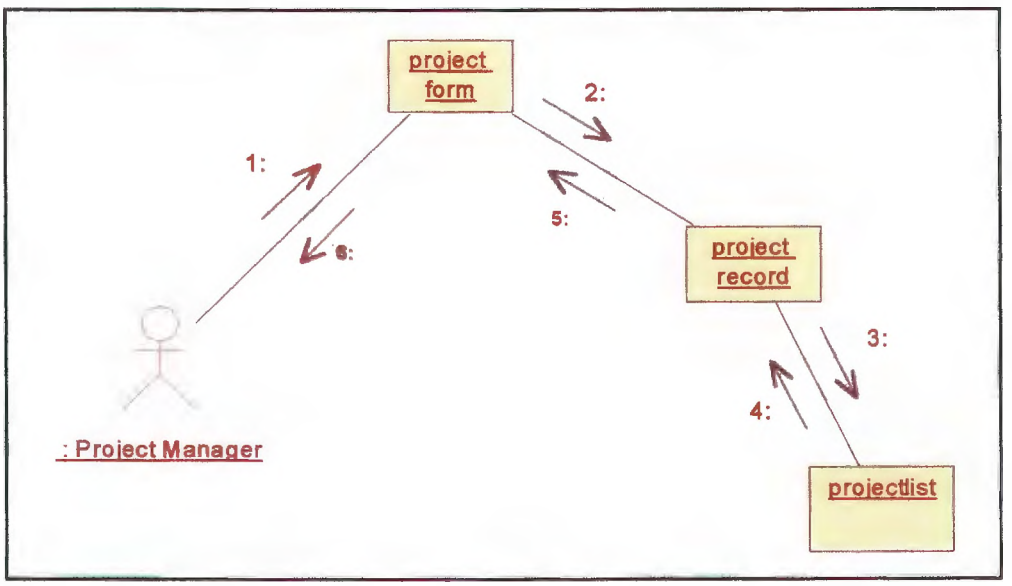


Figure 6.3 Collaboration Diagram for Create New Project

The collaboration diagrams are similar for both creating a new project and for creating a new employee. Data are collected by the project form which in turn collaborates with the project record object and the project list object in order to complete the task of creating a new project. The same type of collaboration occurs when adding a new employee.

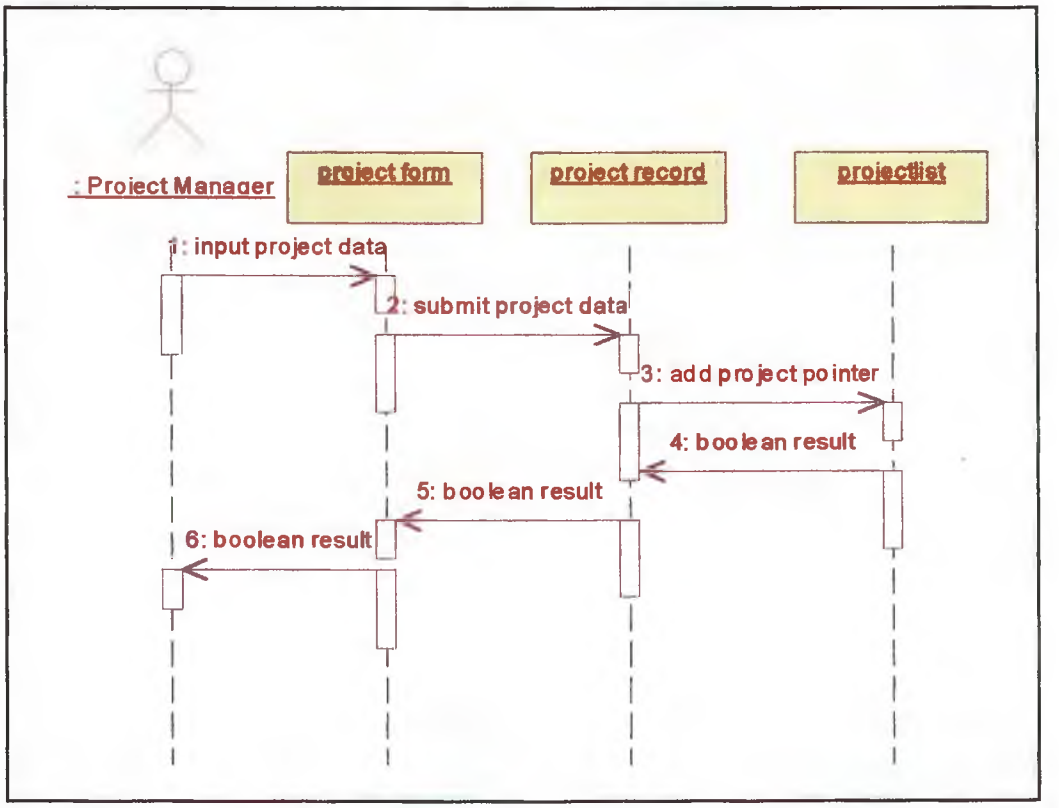


Figure 6.4 Sequence Diagram for Create New Project

Figure 6.4 illustrates the sequence of events for adding of a project by the project manager. Note that the return values are indicated as messages. One of the problems of utilizing this technique of documenting return values is that Rational Rose responds as if these are messages to the object(s) and thereby cannot generate the proper class methods. This inadequacy leads to faults in the generated code. Whenever a CASE tool produces code that either incorrectly represents the design or contains defects then the need for additional examination and testing of

the software is recognized. Weakness in a development tool is yet another potential source of defects. However, even though this is a concern it is outside of the scope of the research presented in this thesis.

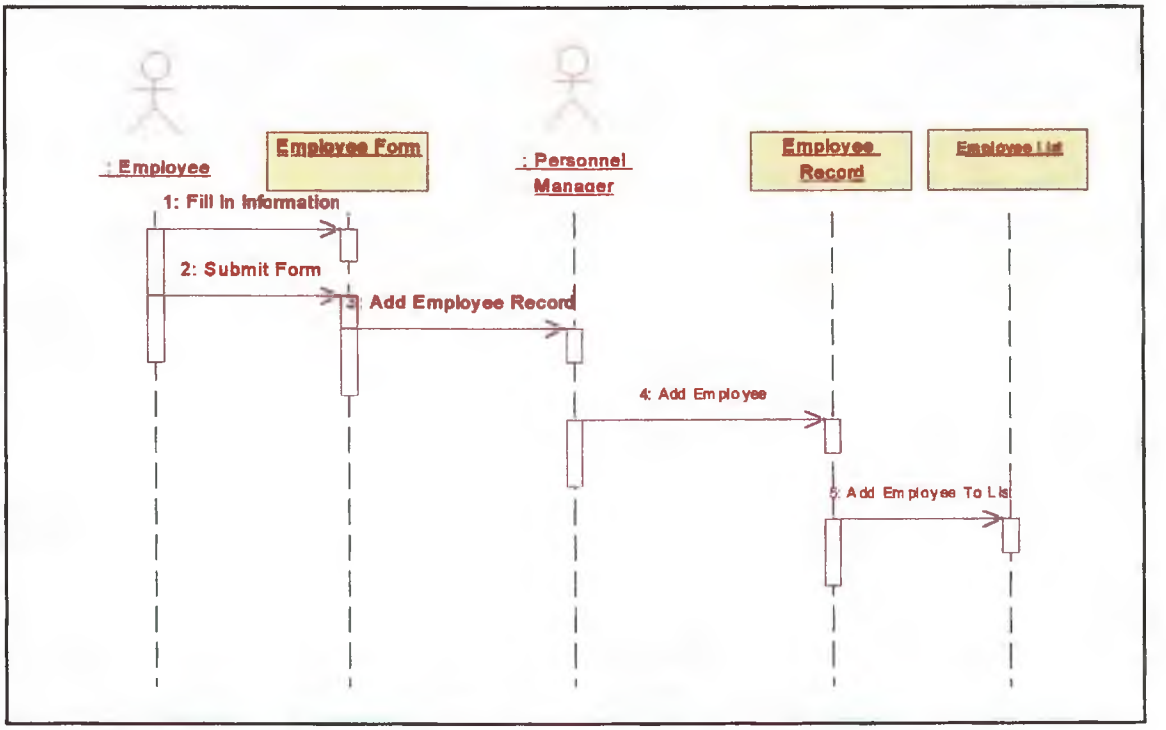


Figure 6.5 Sequence Diagram for Create New Employee

In this instance, Figure 6.5, the return values were not indicated; however, one will note that there are two actors involved in this sequence diagram. First the Employee completes an employee form and then submits that form. The Personnel Manager then accepts that form and adds the employee to the employee record object and the employee record sends a message to the employee list to add it to the employee list.

Once the sequence and collaboration diagrams are developed then one uses the information

contained in each of the diagrams to develop the classes necessary to support each of the objects identified in each scenario. The functionality of each object is identified from its role in the behavioral model. This functionality leads to the development of the class methods. Class attributes are also suggested by information obtained during the analysis phase where one determines the required attributes. In the case of an employee object it was determined that the following attributes were needed.

Table 6.2 Employee Class - Attributes

Attribute	Type	Length
ID	Unsigned Integer	
Name	String	50
Date of Hire	Date	
Position	String	25

For the purposes of the prototype, as one can see in Table 6.2 that the employee class is quite restricted since one is not concerned with any additional personal data outside of the context of the project management system. In a more complete example, one would have additional data collected.

The following diagram, Figure 6.6, presents the entire class diagram for the project management system. The diagram includes all of the classes used to support the objects participating in the project management system. In addition to presenting all of the classes, the relationships among the various class are illustrated. This relational information is used by the CASE tool to generate the class skeletons along with their inheritance and associations. The programmers can also use this diagram to verify that they are implementing the various classes according to the design.

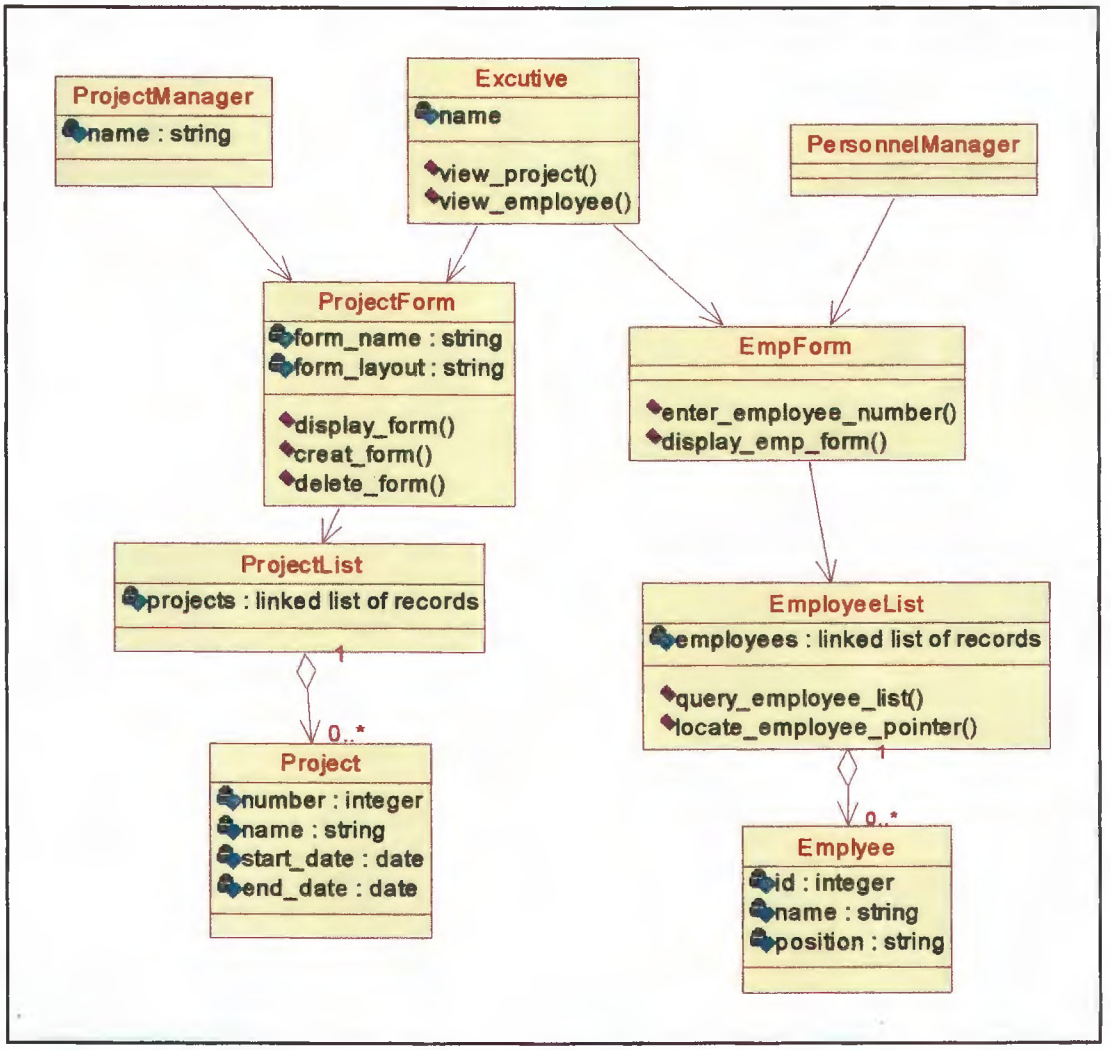


Figure 6.6 Class Diagram for Project Management System

From these initial diagrams the software was developed. Then each of the scenarios discussed in Section 5.6 was simulated. The following sections present the results of how each of the defect introducing scenarios was modeled using the integration testing technique.

6.3 Scenario Testing

In order to demonstrate the effectiveness of the integration testing technique each of the defect introducing scenarios described in Section 5.6 was simulated using the project management prototype. The results of these scenarios are discussed in the following sections. Additional defect examples are provided in this chapter which allow expanded insight into how defects are introduced and how the technique reacts under those situations.

6.3.1 Scenario 1 - Design is Correct, Programmers Introduce Defects

In this instance the design is complete. Under Rational Rose, the CASE tool is capable of generating code for attributes and method signatures of classes. The rest of the code is left up to the programmer. As programmers add code to fill out individual class methods, defects may be introduced, as well as, individual method signatures may also be altered. Depending upon the amount of detail presented in the class diagrams, programmers may be allowed adequate freedom in completing the class methods to introduce defects, even though the design properly represents the requirements.

As can be inferred from the design of the project list, Figure 6.6, the *add_project* method is required to handle the addition of a new project to the *project_list* via the provision of a pointer to the new project and the return of a boolean value indicating whether the addition was successful or not.

Utilizing Rational Rose C++ 98i to generate the C++ code for the project management prototype allowed for the modification of the code by the author to simulate programmer introduced defects. Reverse engineering was then performed on the code to reproduce the design. This was followed by a comparison of the pre- and post-implementation diagrams, namely Figures 6.7 and 6.8 respectively.

The class diagram, as one can see in Figure 6.7, does not provide detailed information about how the actual class methods are implemented. Only if the activity diagrams are included will the actual control flow for each method be presented. It is common practice not to develop such detail unless there is need for information about a complex algorithm. Only the method signatures are provided in the class diagrams. Additional information can be collected and stored in the class definitions under Rational Rose. To access that information one must *click* on a particular class's method for the actual formal parameters to be revealed. In this instance clicking on the add_project() would reveal that a pointer was to be used when inserting a new project to the linked list. Furthermore, unless there is sufficient documentation, the programmer might not understand that a linked list is to be used and an array may be employed instead. Still, a pointer would be indicated. The accidental use of a project's id in place of a pointer would result in a run time error that may not have been identified until the code is executed.

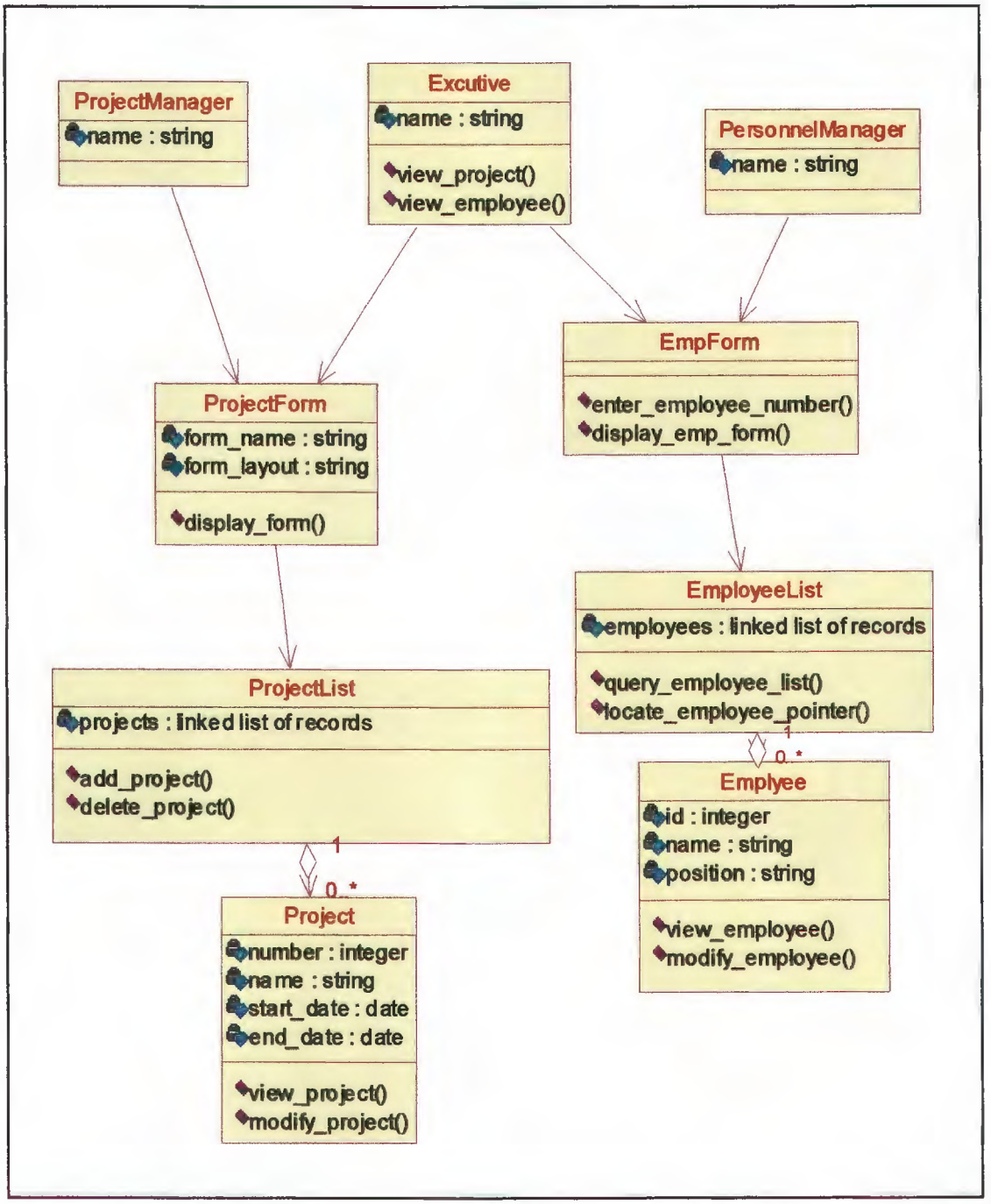


Figure 6.7 Pre-Implementation Class Relationship Diagram

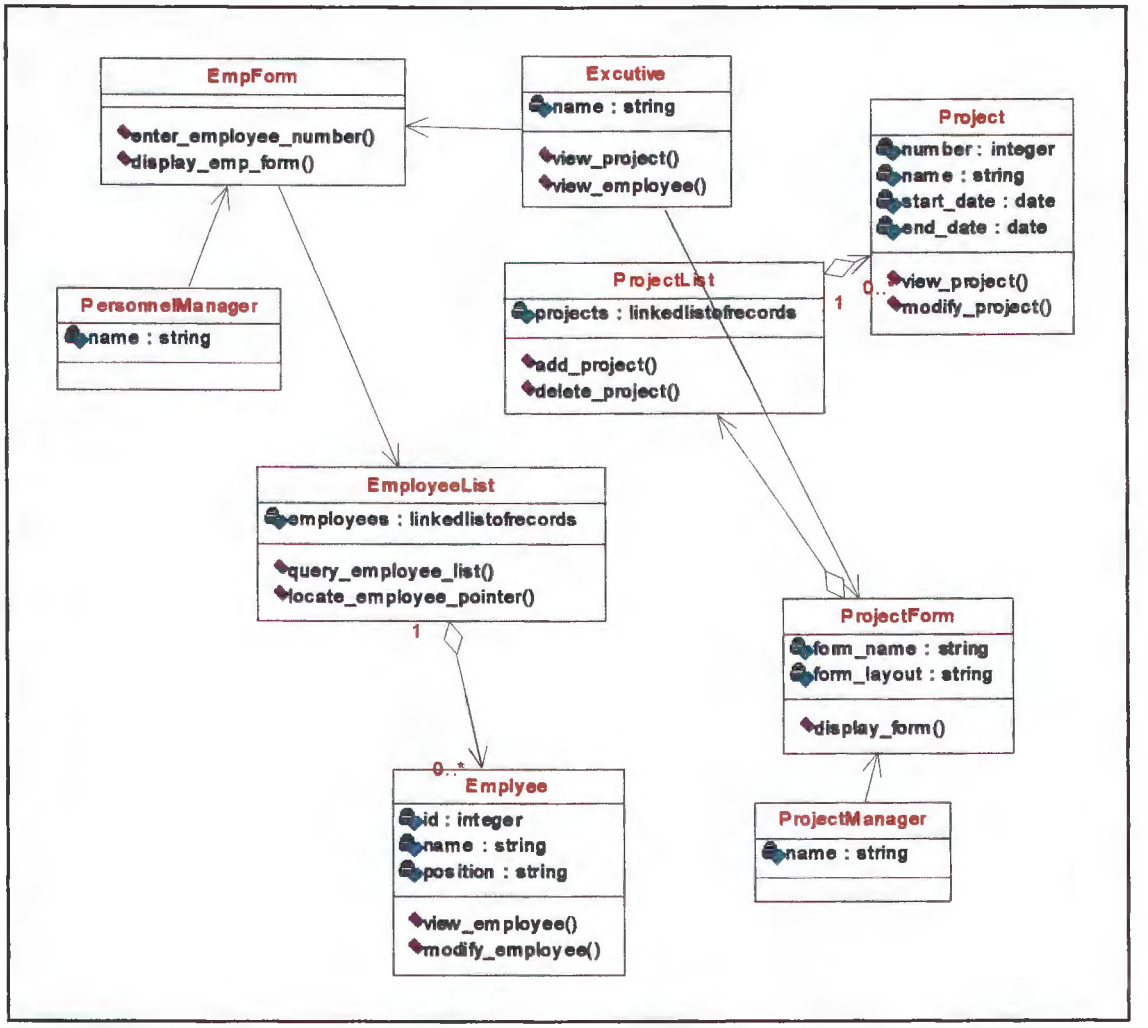


Figure 6.8 Post-Implementation Class Relationship Diagram

As can be seen in Figure 6.8, the reverse engineering tool of Rational Rose C++ 98i created a diagram that is similar to the one built by using the design tools. Again, one is not able to distinguish the differences in the two signatures for the class method - *add_project()*. Sufficient information is not provided in the diagrams produced by the printing function of Rational Rose. Even employing an extended view of the method, as provided by Rational Rose, did not reveal the correct parameters for the method. The tool, in an attempt to understand the use of the * as indicator of a pointer, converted it to an X which does not correspond to proper C++ syntax:

add_project(projectXproject_addr:void):boolean. Here one can see that the return type for the add_project method is correct, *boolean*; however, the formal parameter does not represent a pointer to a project. In this instance there were two separate defects - the simulated introduction of defects into the implementation and the inability of Rational Rose's reverse engineering tool to properly reverse engineer the design from the source code.

Still, with this information, one is able to determine that the design and the implementation do not correspond. Further examination of the source code reveals that the implementation was flawed. The design was correct, however, it was simulated that design diagrams were misinterpreted thus introducing defects into the code.

6.3.2 Scenario 2 - Design is Incorrect, Programmers Follow Design

This scenario is much more difficult to illustrate within the context of the prototype. Defects may be introduced into a software system for a variety of different reasons ranging from mis-communication between the user and the analyst to inexperience in the use of automated software design tools. It is not uncommon for the lack of detail or incomplete system requirements to adversely affect the design of a system. Utilizing a CASE tool like Rational Rose will certainly aid the analyst in designing a system that is at least logically consistent.

Rational Rose provides a tool for checking a model which in return produces a log of all errors. The nature of Rational Rose, and its weakness in allowing for modeling *return values* in sequence diagrams, does reduce the effectiveness of the tool in correctly identifying defects within a design. Employing the model checking capability, along with the integration testing technique of this thesis, will certainly improve the quality of a design. Correctness of a design is far more elusive since a complete, though flawed design, may actually pass the design checking process.

Under this scenario, the pre- and post-implementation diagrams may very well correspond.

However, as the design is implemented, either through the use of a CASE tool or by programmer coding, there will be continued review of the UML diagrams representing the design and the implementation. Here the test group will have the opportunity to examine these diagrams and possibly identify the faults in the design and/or implementation. This opportunity, if the software project is properly managed, will occur throughout the life cycle. Because of the incorrectness of the design there may be instances during the life cycle where the design will have to be modified.

The same problems as in the case of Scenario 1 may occur, but in this case the design that resulted (even as it was adapted earlier) is incorrect. The implementation will result in incorrect results (output). Reverse engineering the code and comparing the pre- and post-implementation diagrams may reveal no difference (programmers implemented the incorrect design), or may reveal differences as in the case of Scenario 1.

As with the other scenarios which involved incorrect or incomplete designs, the technique will help in highlighting instances in the implementation where programmer modifications were made to the design. The defects showing up as the software executes may be due to the defects in design logic (the software was implemented according to the design and corresponds to it, but the result is incorrect) and not due to the differences between the design and the implementation. In this scenario the pre- and post-implementation diagrams may be equivalent. It may be later in the integration and system testing that the defects are recognized. It is only hoped that with the additional review and inspection, brought on by the use of the diagram comparison element of the technique, that the testers will recognize the faults in the design prior to completion of the implementation.

To simulate this scenario the project management system was modified to provide for a defective design. The method *view_employee()* was altered so that it returned additional data, other than the data that should have been requested in the design. This defect, simple in concept, illustrates

the basic fundamentals of a flawed design. The implementation followed the design and did as was specified and returned the same data elements.

Figures 6.9 and 6.10 illustrate the method signatures for the *view_employee()* method. As can been seen, the two signatures are the same. Even though they correspond, the design is flawed since additional data is begin returned for the employee. In a more complete and perhaps, more data sensitive system, for example a payroll system, one could have inadvertently returned personal data. Only after the system is undergoing system testing and end-user testing would the defect possibly be discovered.

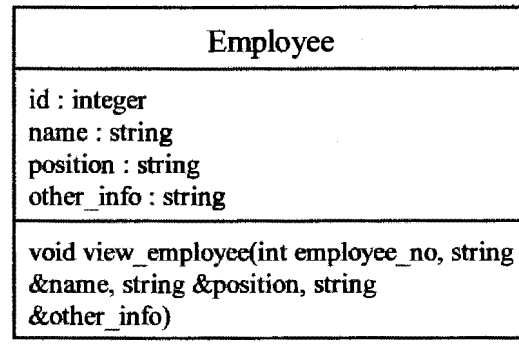


Figure 6.9 Pre-implementation Class Diagram for Employee

As can be seen, the employee class contains four different attributes - *id, name, position,* and *other_info*. When the *view_employee* method was designed all four data elements were included in the arguments to the method. Note that references were used to return the data from the function, thus allowing for more than one value to be returned.

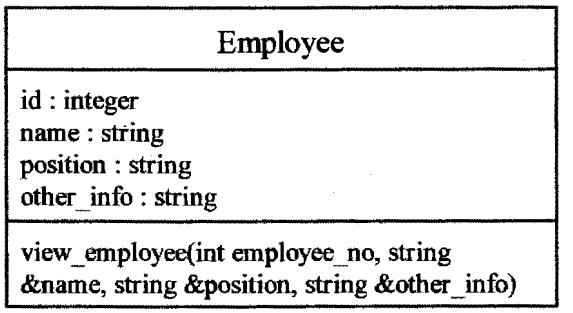


Figure 6.10 Post-Implementation Class Diagram for Employee

The post-implementation diagram illustrates that the design was followed. Again, all of the data elements were returned, not just a selected subset.

In this simple illustration it might not matter that all of the data elements were returned. In fact, when the data is displayed the design may very well not call for displaying the *other_info*. Still, this is a defect in the design which was implemented in the software. Even as the implementation continues it is possible that the implementation will continue to contain defects which are a result of an incorrect design. Unless those design faults impact on the functionality of the system they may not be found until the system is complete and operational.

6.3.3 Scenario 3 - Design is Incorrect, Programmers Attempt To Correct It

As in Scenario 1, where the design is correct and the programmers introduce defects, the pre- and post-implementation diagrams may differ, though those differences will depend upon the type of defect present. Defects in the method signatures can be detected through the use of detailed class diagrams which provide adequate information about the methods, their parameters, and return values. Defects in object interaction, e.g., modified object sequencing and object collaboration, can be identified through the use of sequence and collaboration diagrams.

In order to illustrate the testing technique, the design of the project management prototype was

arbitrarily modified by the author to allow for missing elements. Here the sequence diagram for adding a new project, see Figure 6.11, was modified to exclude certain elements.

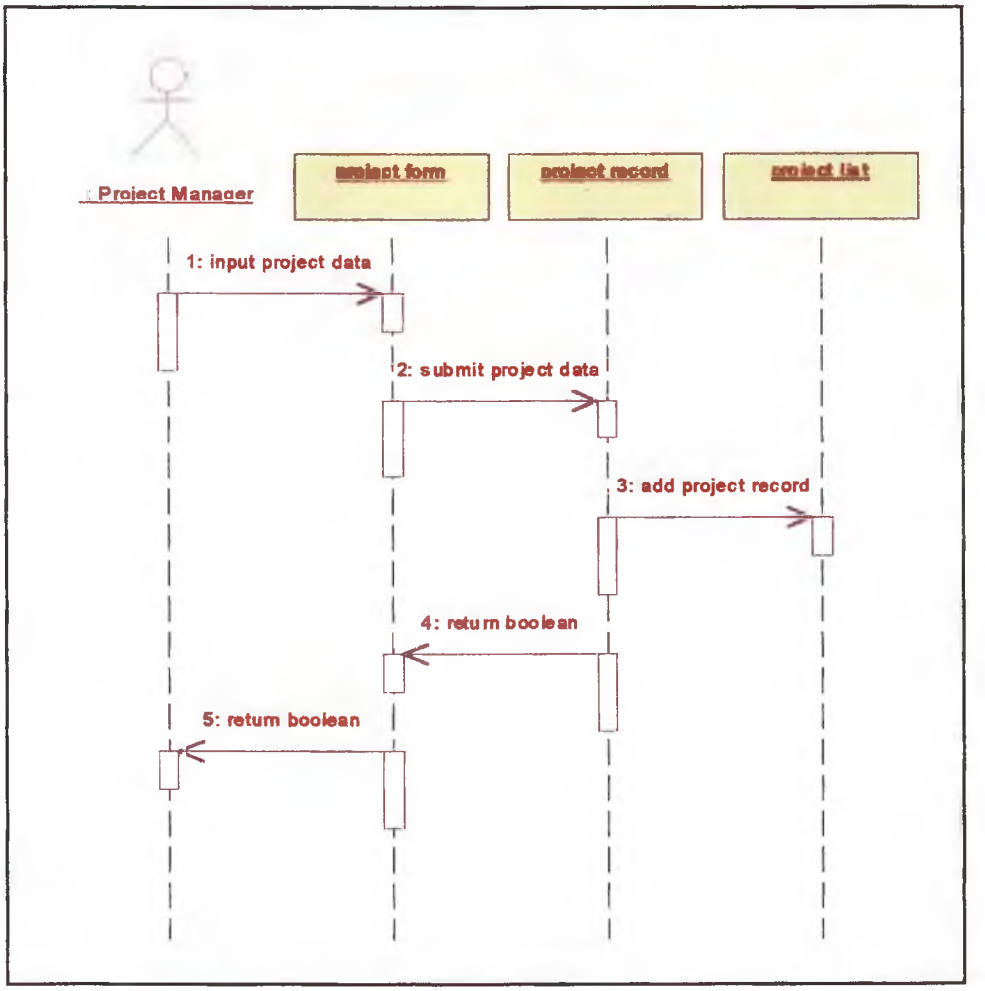


Figure 6.11 Pre-Implementation Sequence Diagram for Creating Project - Scenario 3

This sequence diagram, Figure 6.11, illustrates how a project record is added to the project list, however, exactly what data is stored in the project list remains vague. Furthermore, the fact that a boolean value should be returned indicating success or failure of the insertion into the list was intentionally left out of the sequence diagram to simulate a design defect.

In simulating the correcting of the design, the project record was determined to be a linked list of records, not a linked list of pointers, and implemented as such. The implementation was simulated so that there should be a boolean return value indicating success or failure of the record insertion. Figure 6.12 presents the sequence diagram representing the implementation.

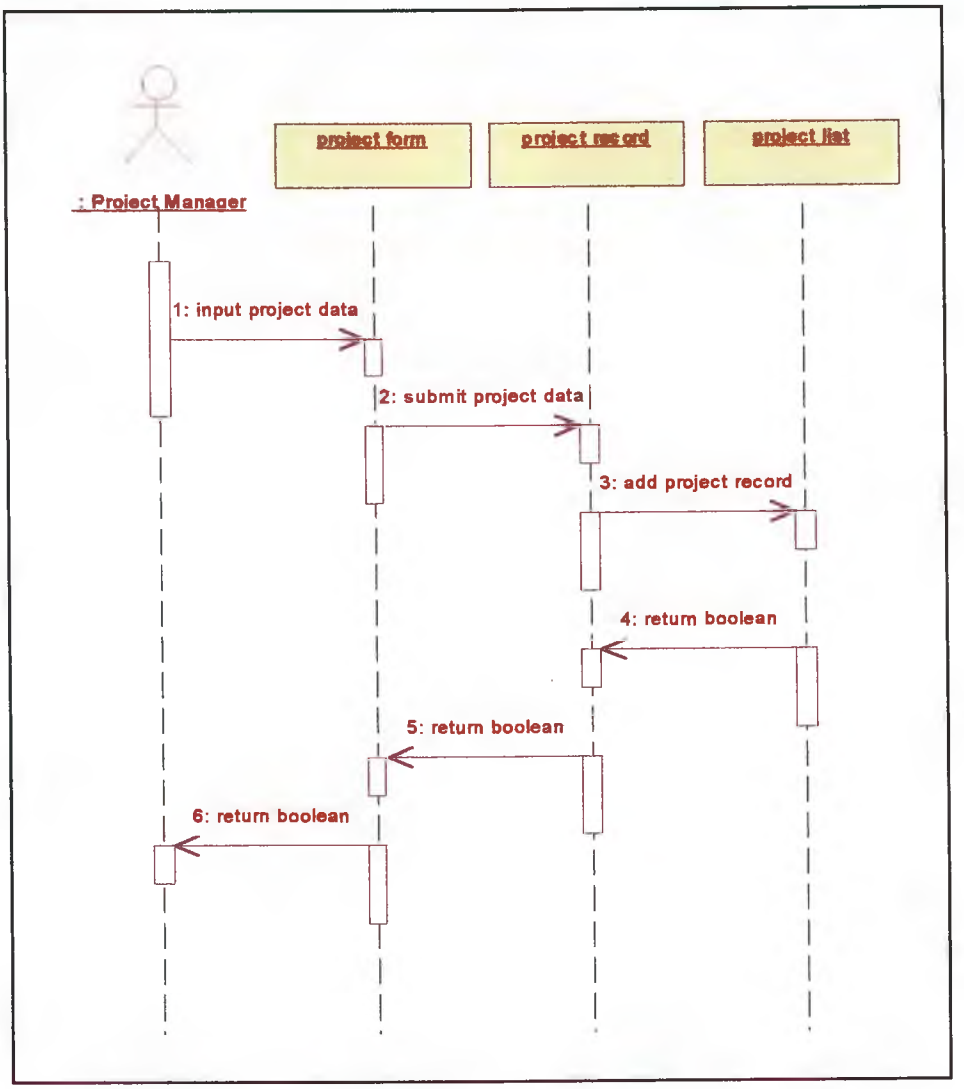


Figure 6.12 Post-Implementation Sequence Diagram for Creating Project - Scenario 3

By examining Figures 6.11 and 6.12 it is apparent that there is a difference between the diagram

representing the design and the one representing the implementation. Highlighting those differences leads to the need for reexamining those places which indicate a fundamental difference. The issue of not returning a boolean value indicating success/failure from insertion into the *project list* is easily determined as a defect in the design. Making changes to the design and then regenerating the pre-implementation sequence diagram will bring those two diagrams into agreement in regard to that point. However, the failure of the design to correctly indicate that the *project list* is to be a linked list of pointers was not corrected in the implementation simulation. Therefore, the implementation remained flawed, even though in this instance the sequence diagrams do correspond with one another, save for the boolean return value being missing from the pre-implementation sequence diagram (see Figure 6.11). Only after further examination of the sequence diagrams was this defect detected. If this defect was not identified, it is conceivable that the linked list would have remained in the implementation as a linked list of records, not pointers. The software may have functioned correctly, however, if by chance, other portions of the design, such as a reporting or query function, did have a more complete design that correctly indicated that the *project list* was a linked list of pointers, then another fault would have occurred.

From this scenario it is evident that no one portion of the design can be isolated and compared with the corresponding implementation. One has to take a consolidated approach to integration testing. Using the pre- and post-implementation diagrams one should examine all common elements across all of the available UML diagrams in order to isolate potential defects.

6.3.4 - Scenario 4 - Design is Correct, However Lacking Detail, Defects Occur

This scenario correlates closely with Scenario 3 in that defects are introduced both from the design and by the programmers. Due to the lack of detail, the simulation is left with little choice except to expand the design during implementation, making decisions where the design is lacking. This scenario is perhaps the most common one encountered in object-oriented development. The design may adequately represent the system's requirements, however, details are missing which

allow for the introduction of defects as the code is developed. Programmers, left with the responsibility of completing design details, may make decisions which impact negatively on the correctness of the implementation.

Rational Rose's tool for generating C++ code from the design focuses its attention on the class diagrams. Code is generated which represents the classes and their relationships; however, the sequence and collaboration diagrams are not converted to C++. Furthermore, the individual class methods are not produced by the tool. This lack of detail places a burden on the designer to fully document the system including all necessary information which can be used by the programmers to develop the actual code.

It is often the practice to omit the design of a class' methods unless there is a complex algorithm being employed. In other words, the activity diagrams of UML are not created during the design phase. Furthermore, system designers often leave the detailed implementation decisions to the programmers. Those areas of the design which are left undocumented can lead to flaws in the implementation.

In order to simulate the technique under this scenario, the design for inquiring on a particular employee was left incomplete. Instead of explicitly stating that the container class was a linked list of employee record pointers, the design illustrated that there was a linked list of employee records. It was simulated that in developing the inquiry method there was adequate information indicating the use of the linked list of pointers. Therefore, the design was correct, though incomplete. The pre- and post-implementation diagrams, both the sequence and class diagrams, provide adequate information pointing to the mismatched data types.

With the use of the technique of comparing pre- and post-implementation diagrams it was possible to identify the presence of a defect in the implementation. Again, as in Scenario 3, the sequence diagrams greatly assisted in indicating there was a difference between two elements of

the design. Further examination of the class diagrams pointed to the fact that the *query_employee* and *add_employee* member functions used different arguments, one an *employee_id* and another a pointer to an employee record. Highlighting these differences lead to the discovery of the defect in the implementation. The initial design failed to indicate that the *employee_list* was to be a linked list of pointers allowing non-corresponding implementations.

The *EmployeeList* has the following design, see Figure 6.13. Here one can see that the *LinkedList* does not indicate whether it is to be a list of pointers to employee records or a list of employee records.

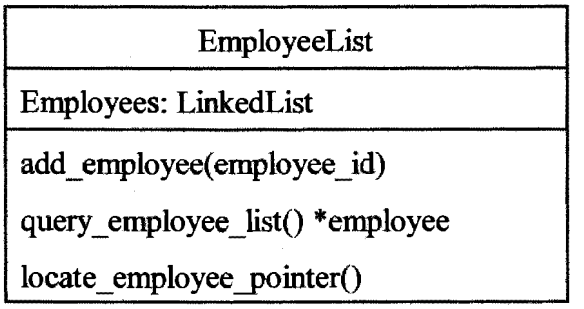


Figure 6.13 Pre-Implementation EmployeeList Class Diagram

Because the information is lacking in the design a decision had to be made regarding the implementation of *query_employee_list()*. Since the specification called for a return value of a pointer to employee then it was implemented using a routine which accepted an employee id number and returned the pointer to that record, or a NULL if the employee id was not matched.

EmployeeList
Employees: LinkedList
add_employee(employee_id) query_employee_list(Employees: LinkedList) *employee locate_employee_pointer()

Figure 6.14 Post-Implementation EmployeeList Class Diagram

Additional detail of the implementation would have to be found through the use of an activity diagram which illustrate the use of the linked list of records and the fact that a NULL would be returned in case of failure to discover a matching employee record.

With the subtle change in the design, the use of the comparison of pre- and post-implementation diagrams would be helpful in at least determining that there was a difference in the detail for the *query_employee_list* method.

6.4 Additional Defect Types and Integration Testing Technique

The four scenarios, given in Table 5.5 and simulated in this prototype, were limited in their ability to demonstrate the integration defects outlined in Table 5.6. Therefore, two additional defect examples were developed using the project management system. The examples were used to demonstrate the technique with object state defects and incorrect object collaboration.

6.4.1 Object State Defects

To use the state diagrams properly for software testing, it is essential that the diagram include both the initial and the resulting state of an object after it has been modified. Presenting this

information in a state diagram allows the user to compare those values with the message sent to the object and check the resulting state of the object after the action requested by the message has been completed. Note, following are a number of reasons for which an object can have an invalid state:

1. Improper message,
2. Incorrect external interface, and
3. Internal method flaw.

Tracking down the cause of the defect can be challenging, however, the intent of the technique presented in this thesis is to assist the tester in locating flaws in the software. The actual correcting of the software will be aided by use of these diagrams, still, further investigation and use of additional test runs with controlled data may be required to actually pinpoint the location of the offending code.

Due to the limited scope of the prototype, the project and employee objects are treated similarly. When the project object is created the state of the new object is undefined. As the data change, the state of the object changes. A project's *end_date* should be greater than or equal to the project's *state_date*. These two dates can be NULL if they have not been determined at the time the project is created. Figures 6.15 and 6.16 illustrate the use of pre- and post-implementation sequence diagrams to determine the cause this defect.

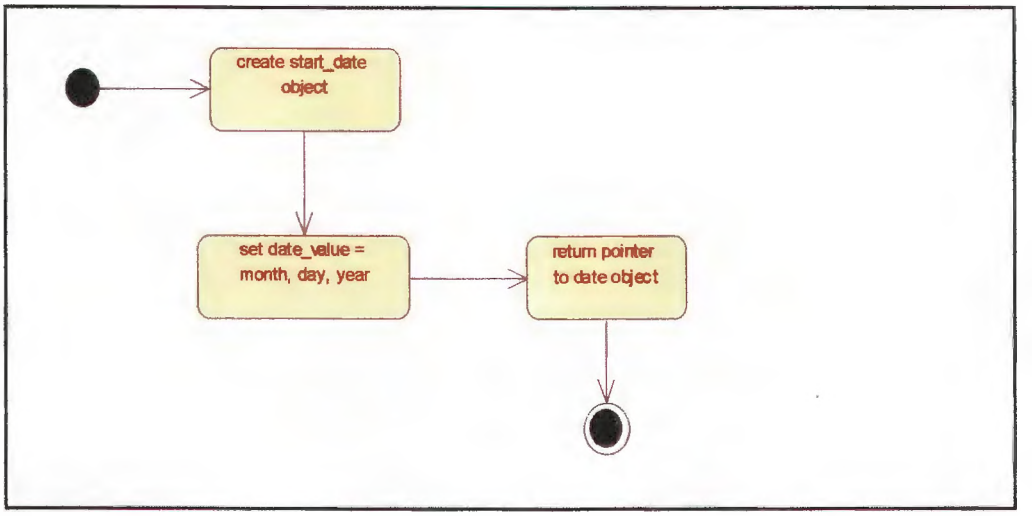


Figure 6.15 Pre-Implementation State Diagram for Set start_date

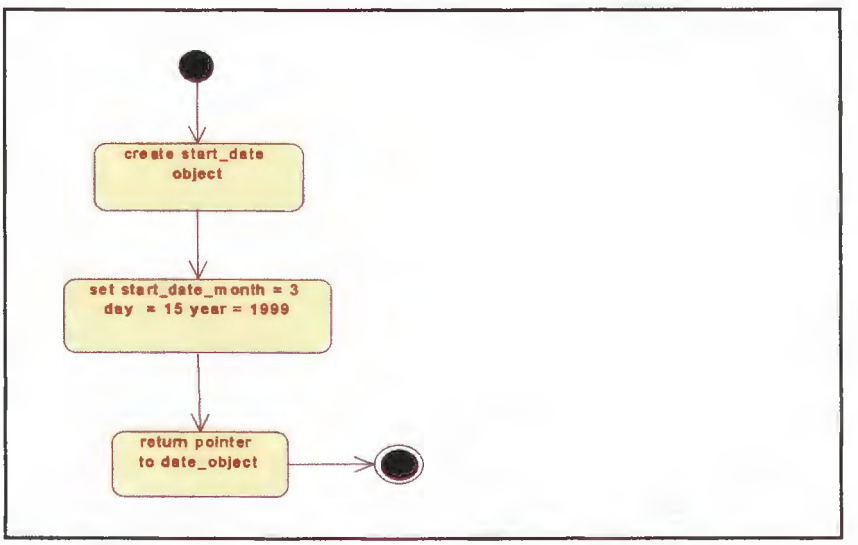


Figure 6.16 Post-Implementation State Diagram for Set start_date

This author recognizes that object-state information relies heavily on the internal mechanics of the

class methods which are traditionally the purview of unit testing. However, the state of an object oftentimes impacts other objects, as well as the behavior of the software, particularly when the state of an object is accessed through communication with other objects.

6.4.2 Incorrect Collaboration of Objects

As discussed in 5.4.1, collaboration diagrams illustrate the cooperation between objects in completing a particular task. In the prototype project, *date* and *linklist* objects collaborate in the creation of a new project, as well as, in the listing of projects already assigned to the management system. Figure 6.17 illustrates the pre-implementation collaboration design for the prototype. Here one is able to see which objects are collaborating.

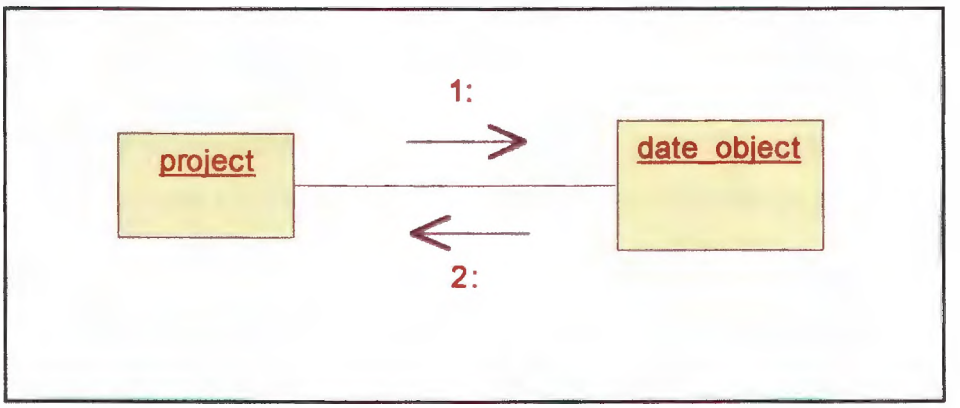


Figure 6.17 Pre-Implementation Collaboration Diagram - Set Project Date

The project object collaborates with the date object in the creation of a new project. The date object is created and then it returns its address. The pointer is then stored in the project object. It takes the collaboration of both of these objects to create a new project.

Figure 6.18 takes data produced from the software and illustrates it via a collaboration diagram.

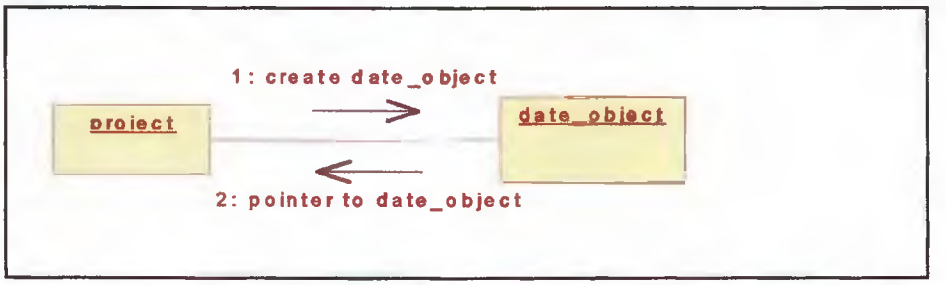


Figure 6.18 Post-Implementation Collaboration Diagram - Set Project Date

The post-implementation diagram contains more detail than the design diagram. The degree of detail is dependent upon the standards and the individual designing the system. [Quatrani, 1998] comments on this very issue. When responding to the question of how complex one should make interaction diagrams he states that the diagrams should be kept simple. These diagrams are designed for providing one with a visual representation of the object, the object interactions, and the messages passed between the objects. In addition, the interaction diagrams represent the functionality of the use case or scenario that they are modeling. However, as illustrated in Figures 6.17 and 6.18, enough information must be included in the design diagrams so that one is certain that they are implementing the system as desired. Limiting design diagrams too much presents serious problems for both development and testing. As one can see, there is an ongoing conflict between providing diagrams simple enough to fully comprehend while at the same time present adequate detail to support the design of complex systems.

6.5 Integration Defect Types and Detection

In order to further demonstrate the integration testing technique, in regard to the integration defects outlined in Table 5.4, the following sections discuss the use of UML diagrams, in particular state and collaboration diagrams, to model object-oriented software and to assist in the identification of such defects within the design and the implementation.

It is recognized that the following examples are quite limited and do not present the technique as it would be used in a complete system development. The discussion, however, is intended to examine the employment of the technique in a limited situation that targets a particular defect type. Under normal conditions the system under development would be much larger and more complex. Occurrences of multiple defects and overlapping causes would be far more prevalent.

As mentioned in Section 6.3, to determine the effectiveness of the technique in defect detection and identification, the software was modified to allow for the introduction of defects that are reflective of the types of predicted defects outlined in Table 5.4. It must be recognized that any particular defect may be the result of one or more of the following defect types. This overlapping of defect types merely reenforces the need for multiple views of the software in order to identify the existence of a defect, and hopefully, assist in its correction.

The following sections simulate tests where the type of defect for which the software was seeded, and the results of a test run are considered. Each of the examples is designed to examine the use of the integration testing techniques in identifying the presence of one particular integration defect type outlined in Table 5. 4.

6.5.1 Defect Type 1 - Improper Inheritance

Even though one could argue that improper inheritance may not be a demonstrable integration type defect, it is noted that one can identify such defects through the use of UML models developed from the source code. Both class and object diagrams reveal the class and object structure, including single and multiple inheritance. Furthermore, defects in class members may well be reflected in integration defects since integration defects focus on the following:

1) interface misuse
2) interface misunderstanding, and
3) timing errors. [Sommerville, 1996].

In object-oriented software, since the object encapsulates both structure and behavior, it is important that one examines both the static and dynamic aspects of classes and objects. Thus, in attempting to find the presence of integration defects, examining inheritance is another important area to examine.

```
class parent {
  private:
        int x;
  public:
        parent( ) {x = 10;};
        ~parent( ) {};
        void display_x( ) { cout << "Parent: display_x( ) x " << x << endl; };
  };

class cousin {
  private:
        int x;
  public:
        cousin( ) {x = 30;};
        ~cousin( ) {};
        void display_x( ) { cout << "Cousin: display_x( ) " << x << endl; };
  };


class child : public cousin {
  private:
        int y;
  public:
        child( ) { y = 20; };
        ~child( ) {};
        void display_y( ) { display_x( ); cout << "Child: display_y( ) y " << y <<
endl;};
  };

void main( )
 {
 parent p_object;
 child c_object;
 p_object.display_x( );
 c_object.display_y( );
```

Example 6.1 Improper Inheritance

This example contains single inheritance. The original design, illustrated by the class diagram in Figure 6.20, called for the child class to inherit from the parent class, when in fact a coding defect occurred, causing the child class to inherit from the cousin class. By the fact the two classes have identically named member functions, an example of function overloading, the program compiled and executed without syntax or runtime faults. The program execution reveals no indication of the logic defect. By comparing the pre- and post-implementation class/object diagrams it was readily apparent that the *cousin* class was used instead of the *parent* class.

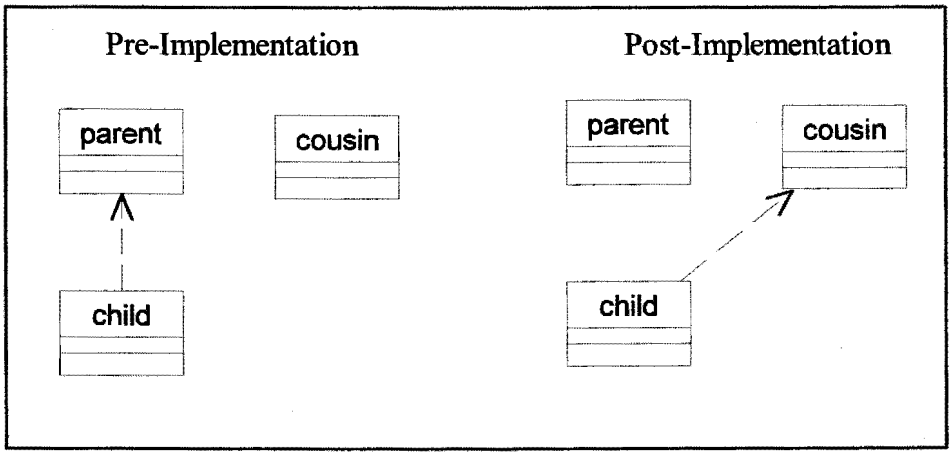


Figure 6.19 Class Diagrams Demonstrating Improper Inheritance

This example is quite elementary, however it was designed to illustrate the concept of using pre- and post-implementation class diagrams. As a software system grows in complexity, it is easier for one to misunderstand the inheritance requirements and from that misunderstanding, attempt to improperly carry out inheritance. This type of defect may result in a situation where the actual code will prevent one from developing software that can be compiled and linked since additional code violations may occur.

6.5.2 Defect Type 2 - Improper Message Sent

Type 2 defects can occur in a number of different ways. In C++ there are features that facilitate the use of member functions but also lend themselves to allowing for defects to be introduced into the software. Additional defect detection and correction are necessary. Specifically, these features are function overloading and default arguments. Function overloading allows one to develop multiple member functions that have the same name. Functions are then differentiated by the difference in arguments. An additional variety of function overloading, though generally not referred to as such, is illustrated in 6.5.1 where the question of improper inheritance is discussed. Here two functions with the same name, and same arguments are allowed to exist.

The use of default arguments is another means by which incorrect message passing can occur. In this instance, arguments are allowed to be set as default arguments in the header of a member function. If the member function is called and there are missing arguments then the default arguments are substituted. In this case, if a message being sent from one object to another is incomplete, the default value, though incorrect, can be automatically substituted for the missing data and the program will not exhibit a runtime defect. Therefore, it is necessary that the defect be detected by an alternative means.

Again the software was modified to allow for the insertion of incorrect message passing. In this instance two separate defects were used:

1) the passing of the incorrect message to the 'Start Date' message, and
2) transposition of the two dates as they were passed to the record storage routine.

These defects in misalignment of the data values are difficult to identify since they require the tester to be aware of what are appropriate values. In this instance the software was not able to determine whether the messages were incorrect or not. With the use of test suites created for the purpose of testing the software, the tester was able to pinpoint defects resulting from the transposition of the values.

Type 2 defects can occur in a number of different ways. In C++ there are features that facilitate the use of member functions but also lend themselves to allowing for defects to be introduced into the software. Additional defect detection and correction are necessary. Specifically, these features are function overloading and default arguments. Function overloading allows one to develop multiple member functions that have the same name. Functions are then differentiated by the difference in arguments. An additional variety of function overloading, though generally not referred to as such, is illustrated in 6.5.1 where the question of improper inheritance is discussed. Here two functions with the same name, and same arguments are allowed to exist.

The use of default arguments is another means by which incorrect message passing can occur. In this instance, arguments are allowed to be set as default arguments in the header of a member function. If the member function is called and there are missing arguments then the default arguments are substituted. In this case, if a message being sent from one object to another is incomplete, the default value, though incorrect, can be automatically substituted for the missing data and the program will not exhibit a runtime defect. Therefore, it is necessary that the defect be detected by an alternative means.

Again the software was modified to allow for the insertion of incorrect message passing. In this instance two separate defects were used:

1) the passing of the incorrect message to the 'Start Date' message, and
2) transposition of the two dates as they were passed to the record storage routine.

These defects in misalignment of the data values are difficult to identify since they require the tester to be aware of what are appropriate values. In this instance the software was not able to determine whether the messages were incorrect or not. With the use of test suites created for the purpose of testing the software, the tester was able to pinpoint defects resulting from the transposition of the values.

The creation of test cases is an additional benefit of creating diagrams representing the implementation. One is able to gain greater insight into how the various objects are actually interacting. In certain instances the design diagrams will be limited in detail thus concealing some object collaboration. The post-implementation diagrams will result in improved design documentation.

Originally the test cases will be developed from the use cases during the analysis phase. As the code is developed, more detailed information will be obtained which will allow for the creation of more specific test suites to be used for integration testing. Testing of the interaction of objects, object interfaces, and object states are an important aspect of integration testing of object-oriented software.

To create the test cases, the interaction between objects is noted. The parameters used in the communications are identified, and the sequence of object communications is documented. Utilizing any additional information concerning argument value limitations, test cases are developed which are then used in traditional integration testing where one object communicates with another object in proper sequence. The argument values are modified and the response of the objects is observed, along with the states of the various objects. Defects are noted whenever the objects respond incorrectly, their states are invalid, or the return messages are not within the acceptable value range.

6.5.3 Defect Type 3 - Incorrect Reply

As with Type 2 defects, Improper Message Sent, discussed above, the software was modified to allow for this defect type. The message returned from the inquiry returned the wrong record. Evidence of this type of defect was difficult to identify since the program did return values for each of the variables within the project object. However, like Defect Types 1 and 2, the use of

controlled test case selection and recording allowed the tester to identify the incorrect message reply.

In an attempt to create more complex defects the return value for the new instruction was arbitrarily set to NULL, simulating Defect Type 3 - incorrect message reply. This modification was not easily detected while testing the project entry. Only when the *project list* was displayed did this defect become apparent.

Internal control and flow of member functions impacts on the external functionality of software. However, integration testing is not used to perform this level of testing. The use of activity diagrams would have assisted in the pinpointing of this defect. The message returned from the faulty member function was correct in that it did return the appropriate data type, however, the value was not acceptable. Increasing internal defect trapping and recovery would have been helpful in identifying this defect. Traditional software testing through the use of exercising the code within a controlled environment was helpful, but eventually the running of a code debugger was necessary to locate the defect within the offending member function.

Due to the weakness of the proposed technique to identify the existence of this type of defect, an additional defect was devised in order to further clarify the ability or inability of the technique to locate Type 3 defects. Changes were made in the software to force it to select the wrong project from the project linked list. When the project number was entered that number was incremented by one, thus placing the object in an incorrect state. If the projects were assigned sequential project numbers, in certain instances, no data would be returned. With the randomness of the defect in finding and not finding data, as well as, the reporting of data, it was difficult for the tester to comprehend the nature of the actual defect. Still, the technique was able to identify that a defect existed, but not able to tell why.

6.5.4 Defect Type 4 - Improper Object State

This type of defect is more difficult to identify. There are a number of different ways in which the incorrect state of an object can be set. To determine when a defect occurs one has to have a definition of what is an improper object state. Proper defect identification and control should be established within the software which would prevent an object from obtaining a value outside of acceptable bounds. This type of defect prevention should be the responsibility of the internal activity of member functions. Such defect detection should have been tested during unit testing. For all practical purposes, this type of testing is outside of the parameters of integration testing. This fact does not remove the possibility that such defects may be ignored until they are identified during the integration phase.

A second type of defect of this type may result from an incorrect message being sent to an object which places the object in an improper state. Again this incorrect message may be identified by testing for Type 2 defects. In addition, a Type 6 defect, incorrect external interface, can result in an improper object state. Refer to 6.5.6 for an example of such a situation. Still, it must be recognized that not all defects of this type may be identified by looking for incorrect messages. Through the use of state diagrams, one of the UML diagram types, one is able to locate that point where a particular value went out of range or became unacceptable.

6.5.5 Defect Type 5 - Incorrect Timing

In non-real-time software the timing of events is not critical. Events occur in an asynchronous mode and are often a factor of the speed of the person using the software, e.g., how fast and how often the user requests data input and inquiry. In this demonstration project the system has no dependency upon time. Therefore, it is not possible to test for incorrect timing. Somewhat indirectly related is the testing for proper sequence of events - Defect Type 7. It was possible to simulate this type of defect and to test for its existence. Still, in that instance one is not concerned about timing issues. Therefore, it is not possible to draw a conclusion about the

style: 1) the use of a parameter with the same name as one of the object's data attributes and 2) the internal assignment of a value where an argument is being passed to that attribute. Still, these types of defects do exist in real world software and they must be discovered during testing. This defect in the code did not cause the program to abort, it simply allowed the constructor to incorrectly set *x* to an improper state.

This type of defect is difficult to find with traditional testing since the program does compile and execute without any visible defects. Depending upon the value of *x*, incorrectly obtained from uninitialized memory, the program could pass testing.

This defect is a concern of unit testing. However, the internal structure of the constructor does not directly reveal the defect. Since constructors do not return values, it is assumed that *x*, in this case, was properly set to 10. The *cout* statement, included in the constructor, revealed that value. The only problem is that *x* is the local variable, not the data attribute belonging to the object *x_test*.

This defect, therefore, lies somewhat outside of the limited testing of a single member function and in the realm of unit or class testing. Still, because of the nature of C++ and object-oriented software, inter-method collaboration may need to use integration testing to locate such flaws.

A quick examination of the class diagram does not reveal the defect.

effectiveness of the technique in identifying the existence of timing defects.

6.5.6 Defect Type 6 - Incorrect External Interface

This type of defect is a result of not developing the proper interface for an object. In C++ the proper external interfaces are presented through the *public* member functions. This type of defect is related to improper message sending/replying. However, the defect actually resides in the formal parameters contained in the member functions. The following example illustrates the possible defect that can occur in the member function interface.

```
#include <iostream.h>
class parent {
    private:
        int x;
    public:
        parent(int x) {x=10;cout << "Parent: " << "constructor" << " x = " << x << endl};
        void display_x() { "Parent: " << "display_x " << " x = " << x << endl;};
    };

void main()
  {
    int a_value = 10;
    parent x_test(a_value);
    x_test.display();
  }
```

Example 6.2 Incorrect External Interface

The above example illustrates a simple case of passing an argument, *a_value*, to the constructor for *x_test*. In coding the constructor, *x* was inadvertently also used as a formal parameter and private class attribute. When *x* was assigned the value 10, it was the local variable, e.g., the formal parameter to the constructor was modified, not the class data attribute X. When the member function *display* was called, the value of *x* was uninitialized and the resulting value displayed was incorrect. This program contains two separate defects or examples of poor coding

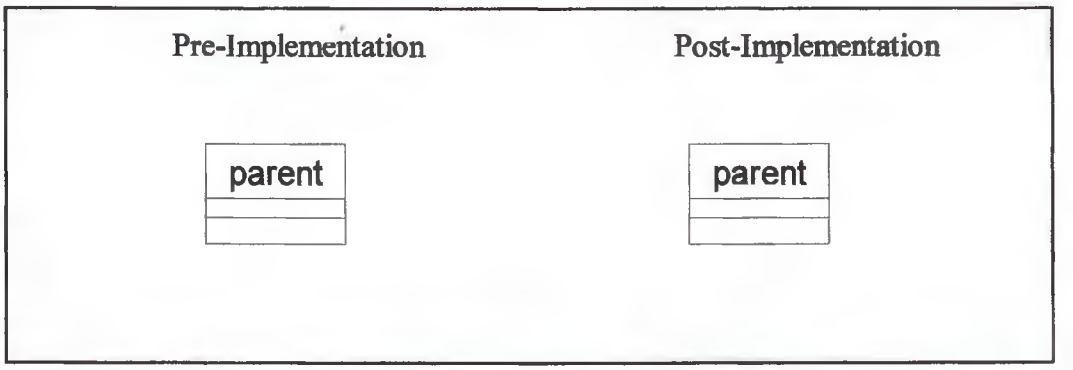


Figure 6.20 Class Diagram Comparison

These two class diagrams shown in Figure 6.20 are identical.

Enhancing the class diagram to include object information provides additional insight in that it includes values and arguments.

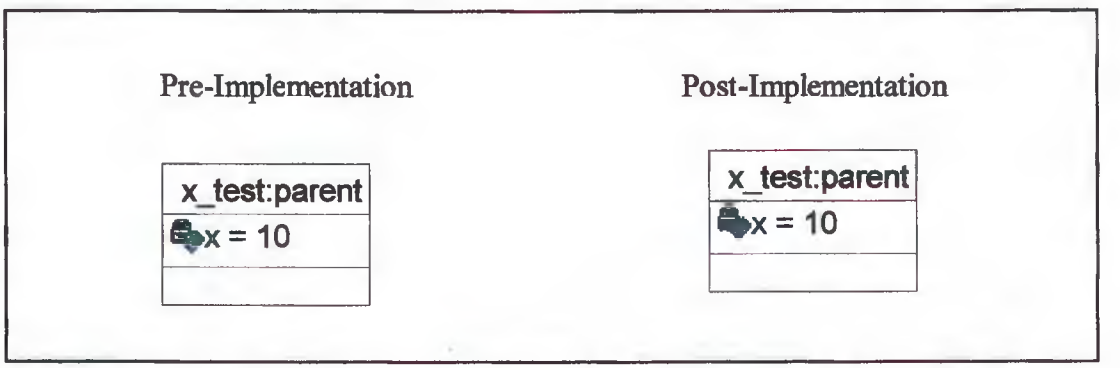


Figure 6.21 Class Diagram With Object Information Comparison

Again the two class diagrams in Figure 6.21 are equivalent.

Depending upon the degree of detail illustrated in an activity diagram, the pre-implementation

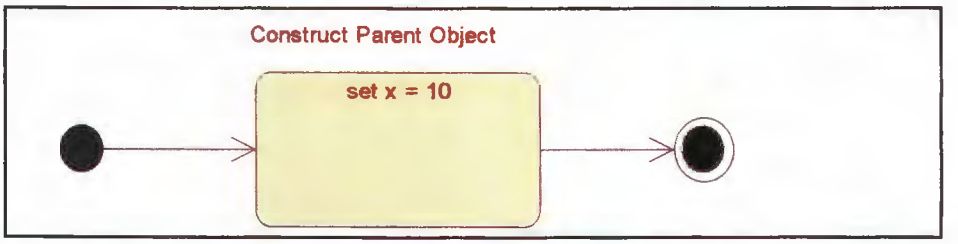


Figure 6.22 Pre-Implementation Activity Diagram

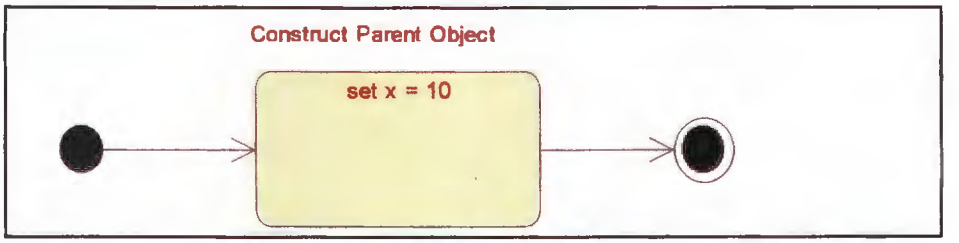


Figure 6.23 Post-Implementation Activity Diagram

Due to the nature of constructors in C++, the argument was indeed passed to the constructor with a value of 10. That argument, however, had the same name as the member data attribute. This fact allowed the compiler to create a separate local variable which was assigned the value leaving the data attribute uninitialized. This flaw is simply an example of scope resolution where a variable declared in a member function that has the same name as a global variable or a variable with broader scope overrides the visibility of the other variable. Only the local variable is visible when the member function is active.

Modifying the parameter's name in the constructor resolved the defect, though, the defect was undetected in the constructor and remained hidden until *display_x*, a separate member function, was called. Use of the compiler's debugger did not point directly to the problem since *x* did have

diagram may give additional information. It must be recognized that the recommended technique for integration testing would not use activity diagrams for the internal examination of member functions. However, when one of the other views does not resolve the defect identification, then one should consider the development of an activity diagram for the member function under examination.

Defect resolution, in this instance, means that the cause of the defect has been determined. When one has an invalid object-state, for example, it may be the result of an incorrect message, an incorrect sequence of events, or even the internal operation of a class method. After examining the external interface of methods and the inter-object collaboration then one may have to investigate the internal construction and behavior of the class methods which impact on an object's state.

Once the presence of a defect has been established through the comparison of pre- and post-implementation diagrams, Figures 6.22 and 6.23, one must then expand the investigation to the cause of the differences between the diagrams. This is accomplished by exercising the software through the use of test cases. These test cases are derived from the use cases and enhanced by the expansion of one's understanding of the program's structure and behavior provided by the post-implementation UML diagrams.

a value of 10 when the debugger was in the constructor. In the *display_x* function, *x* did have an incorrect value. When execution was back in the *main* function, *x* was undefined since it is a *private* data attribute of the *parent* class.

To implement this defect in the demonstration project, a member function was modified in such a manner as to eliminate one of the required parameters. As long as the function call recognized the function, the program would compile and execute without observable defects; however, if one was not cognizant that the function had been modified and that one or more of the parameters were missing, the code could not be compiled and the defect moved from execution time/logic defect to a syntax defect detectable at compile time.

6.5.7 Defect Type 7 - Incorrect Sequence of Events

The software was modified to allow for the violation of the prescribed order of events for a particular instance. In this case the entry of the data elements for a project was placed out of order. Here the Start and End Dates were switched. Even though the software still performed without defect, the order of the data entry violated the design.

Having completed the execution run it was possible to see that the original design called for the entry of the data in the following order:

project number
project name
start date
end date.

The output of the instrumented source code provided the following order:

project number
project name
end date
start date.

From this information it was obvious that the order of message passing deviated from the intended design. Such defect detection is important since, in this case, the switched values were of the same type.

In this instance one is generating a graphical representation of what one would previously classify as a trace generation. However, the use of UML diagrams expands on this approach of analyzing system execution.

6.5.8 Defect Type 8 - Incorrect Collaboration of Objects

Section 6.4.2 presents an example of this defect within the context of the demonstration of concept prototype. As discussed in Section 5.4.1, sequence and collaboration diagrams are used to model the behavioral aspects of object-oriented systems. The diagrams present different views of the same task being performed. It is important that one present enough detail within the diagrams to assure that the software will be properly implemented.

Any modeling tool, unless it is presented with complete and accurate data, can produce diagrams representing the design, which, although logically consistent, may fail to properly represent the system requirements. If that design is used to develop the source code, defects will exist that may not be found directly.

Reverse engineering the design from the software will only produce representative diagrams that equate to the design diagrams. This situation is equivalent to Section 5.6.2 where the design is

defective and the design is implemented.

However, as the software is being implemented, the missing detail may be discovered and changes (enhancements) are made to the implementation. At that point the pre- and post-implementation diagrams will differ, pointing to defects in the design since the diagrams representing the implementation will be more detailed. Defects may also appear as the software executes, not necessarily because of differences between the design and the implementation but as a result of logically defective design.

The following diagrams illustrate the pre- and post-implementation diagrams for this form of incorrect object collaboration.

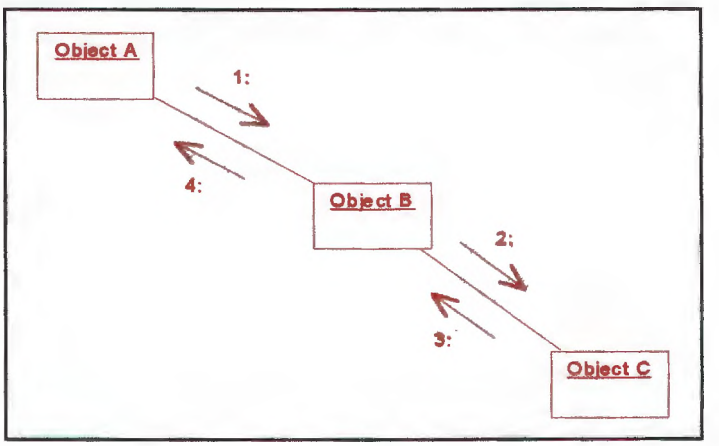


Figure 6.24 Pre-Implementation Collaboration Diagram Illustrating Limited Design Detail

Here the three objects collaborate to perform a given task.

After implementing the system it was discovered by reverse engineering the design that a fourth

object, B' was also involved.

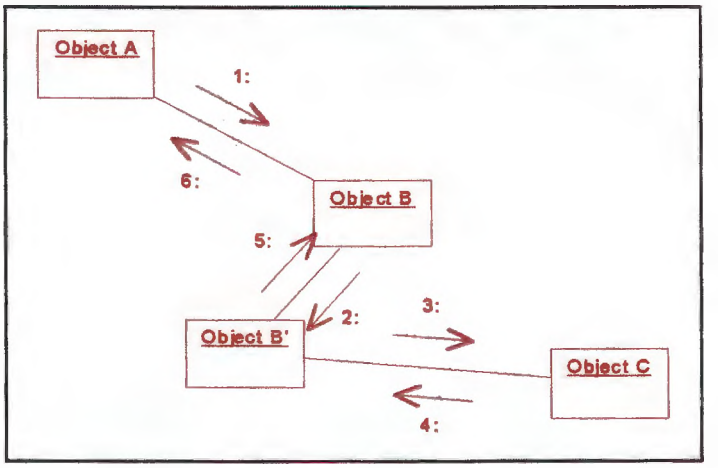


Figure 6.25 Post-Implementation Collaboration Diagram Illustrating Expanded Detail

The differences in Figures 6.24 and 6.25 do not reflect a major defect in the design, just the lack of significant detail to fully implement the task represented by Figure 6.24. This type of defect is an example of Scenario 4, where the design is correct, however, lacking detail, defects may occur. In this instance, one could argue that this example is more attuned to Scenario 3, where the design is defective and the programmers attempt to correct the design.

A second form of incorrect object collaboration can result from an incorrect design. Again logical checking of the design may reveal this defect, however, if the design is consistent then the defect will be passed from design to implementation. Figure 6.26 illustrates an incorrect collaboration diagram.

Hopefully the programmers may recognize this defect where Objects A, C and D are collaborating to complete the task instead of the correct ones, Objects A, B, and C, and make the necessary

changes to the software. Reverse engineering the design from the implementation code will produce diagrams which will lead to the discover of a difference between the diagrams representing the design and those representing the implementation.

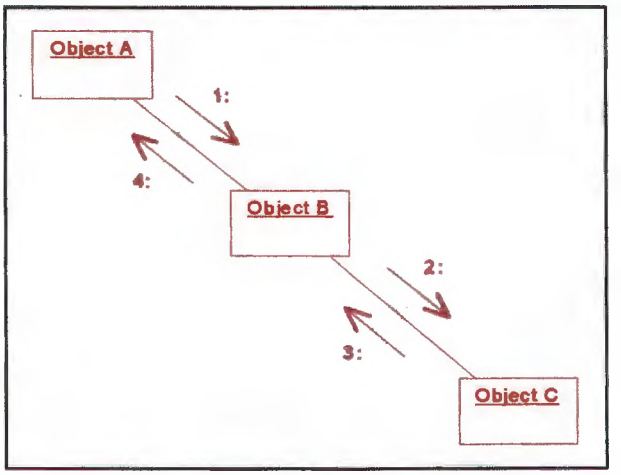


Figure 6.26 Pre-Implementation Collaboration Diagram Illustrating Design Defect

Changes were required to produce the correct response form the software once it was executed. Those changes are reflected in the post-implementation collaboration diagrams shown in Figure 6.27.

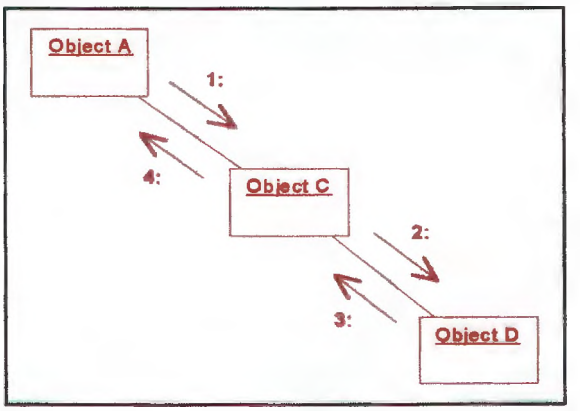


Figure 6.27 - Post-Implementation Collaboration Diagram Illustrating Design Correction

Comparing the two diagrams points to defects in the design.

Different results will occur if the design remains constant and the implementation is altered. Each iteration of diagram creation and comparison will reveal the differences between the object collaboration diagrams. These difference will first assist in determining which model is correct and if the implementation is found to be defective, assist in keeping the implementation on target with the design, hopefully avoiding major revisions of the software once it is released for system testing.

6.6 Summary

This chapter presented the use of the integration testing technique in a number of different simulations. Initially emphasis was placed on the illustration of the four defect introduction scenarios presented in Section 5.6 though the use of a project management system prototype. Each of the defect causing scenarios was simulated with the integration testing technique being employed in an effort to pinpoint defects in the software.

Due to limitations in the scope of the prototype, additional defect scenarios were presented in Sections 6.4 and 6.5. These sections illustrated different defect types as outlined in Table 5.4. Even though the example programs were quite limited in scope, each focused on a particular defect type and the use of the integration testing technique in identifying the presence of defects in the software.

In Chapter 7 the scenarios and code examples in this chapter were used for the purpose of evaluating the effectiveness of the integration testing technique along with pointing out any difficulties encountered in using the technique. Furthermore, any additional research or development found to be needed as a result of the simulations is discussed.

CHAPTER 7

Integration Testing Technique Evaluation

7.1 Introduction

The use of the analysis techniques examined in this thesis provides one with insight into the construction and operation of the software under consideration. Difficulty in understanding object-oriented software is reduced by allowing one to visualize the software through the use of execution tracing. Object-oriented software contains many features, like late-binding and polymorphism, which prohibit one from fully analyzing the structure of the software without observing its internal flow control during execution. As discussed earlier, object-oriented software combines both structure and behavior into individual units known as classes or objects. The interaction of objects establishes the behavior of the software under specific conditions, often

represented as scenarios or use cases.

Much object-oriented software is event-driven. Event-driven software, dependent upon external events such as mouse movement and clicking, does not contain the typical control structure inherent in functional software. Therefore, to understand the internal operation of the software, it is necessary that one have a tool or tools that allows for the viewing of the software as it executes. Multiple executions of the software aid in revealing numerous behavioral permutations of the software.

The techniques suggested in this thesis are aimed at providing quality-control/testing personnel with a means of analyzing software both statically and dynamically, as well as, aiding in the development of sample suites to use in testing the software. The results of the actual use of these techniques are presented in this chapter. Further need for enhancing these tools/techniques and the automation of the technique is also discussed.

7.2 Evaluation of the Technique

The technique provides insight into both the structure and behavior of object-oriented software. The manual aspects of the technique proved to be cumbersome and not easily implemented. The actual software was modified and recompiled to allow for inclusion of the data producing statements. The reverse engineering capability of Rational Rose was used to assist in the creation of the post-implementation class diagrams.

The comparison of pre- and post-implementation diagrams was very helpful in identifying differences between the design and the implementation. Furthermore, comparison of these diagrams led to the discovery of weaknesses in the original design. Since one is never certain when the design is correct or incorrect, it is still necessary to develop a set of test cases which

further exercise the programs looking for integration defects. However, just assuring that the pre- and post-implementation diagrams are equivalent may remove defects which would have resulted in failures during integration and system testing. At least at this point, one can say, with some degree of assurance, the design is indeed implemented. From that point forward one is more concerned with testing the design and less with testing the implementation. Once the design and implementation correspond then many of the defects identified will be a result of a defective design.

With this fact in mind, one then begins by identifying defects, making corrections to the design, modifying the implementation, and again producing UML diagrams and making the comparisons. By following the technique one can save time when completing integration and system testing. More time and effort can then placed on testing the design is to see if it is correct and properly implemented. This type of continuous evaluation and correction will certainly add to the quality of the end product since the proposed technique supplements code reviews and walkthroughs.

To appraise the technique, faults were arbitrarily introduced into the prototype's software. Using known defects made it possible to determine whether the technique was capable of detecting them. The testing of the software against software without known defects was not attempted.

In Table 1.2 evaluation criteria were presented which are an aid in providing insight into the ability of the integration testing technique presented in this thesis to assist in the identification of the presence of defects within object-oriented software. Those criteria were developed in response to the issues raised in Chapter 4 after the review of current research on integration testing techniques. For referential purposes, Table 7.1 reproduces those evaluation criteria.

Table 7.1

Evaluation Criteria

1. Assistance in identifying presence of integration defects in object-oriented software
2. Relationship of technique to dynamic, event-driven aspects of object-oriented software
3. Integration of technique into the object-oriented software development life cycle
4. Scalability of technique to larger projects

The following section applies these evaluation criteria to the integration testing technique. Each of the evaluation criteria focuses on the work reported in Chapter 6 regarding the Project Management System simulation, along with the specific defect type examples in Section 6.4.

7.2.1 Criterion 1 - Assistance in Identifying Presence of Integration Defects in Object-Oriented Software.

The first evaluation criterion is perhaps the most important in determining whether the proposed integration testing technique is able to identify the presence of defects in object-oriented software, and whether or not the defect observed in the implementation is a design or implementation defect.

An attempt was made to simulate each of the defect resulting scenarios discussed in Section 5.6. The types of integration testing defects, outlined in Table 5.4, and correlated with UML diagrams in Table 5.7 were simulated in the prototype in Chapter 6. Additional code examples were presented to illustrate various integration defects.

Table 7.2 Defect Type - Prototype Simulation

Defect Type							
Improper Inheritance	Incorrect Msg. Sent	Improper Object State	Incorrect Msg. Reply	Incorrect Timing	Incorrect External Interface	Incorrect Seq. of Events	Incorrect Collab. Of Object
Scenario 1 Correct Design, Programmer Defects Potential Defects							
X	X	X	X	X	X	X	X
Defects Found in Simulation							
	X				X		
Scenario 2 Incorrect Design, Design Followed Potential Defects							
X	X	X	X	X	X	X	X
Defects Found in Simulation							
Scenario 3 Incorrect Design, Programmers Alter Design Potential Defects							
X	X	X	X	X	X	X	X
Defects Found in Simulation							
	X				X	X	
Scenario 4 Correct Design, Lacking Detail Potential defects							
X	X	X	X	X	X	X	X
Defects Found in Simulation							
	X					X	

As can be seen from Table 7.2, the limited aspect of the demonstration of concept resulted in the type of defect present being limited to Incorrect Message Sent, Incorrect Message Reply, and Incorrect Sequence of Events. However, the supplemental examples presented in Section 6.4 provided additional simulation of the remaining defect types from Table 5.4, save for Incorrect Timing which is outside of the scope of this thesis. Utilizing a more complex simulation would have provided an expanded opportunity for evaluating the effectiveness of the integration testing technique in predicting the remaining defect types. However, the defects simulated in the project management system prototype are representative of the type of integration defects that one would expect to find. Such defects, as incorrect sequence of events and incorrect messages being sent, are the types that are often introduced from the design or by the programmers. Programmer introduced defects, as has been noted, can arise from misinterpretation of the design, the lack of detail in the design or simply coding errors.

7.2.2 Criterion 2 - Relationship of Technique to Dynamic, Event-Driven Aspects of Object-Oriented Software.

The technique, in itself, follows a dynamic process which works with the iterative nature of object-oriented software development. The technique is integrated into the software development life cycle, in particular, the Revised Spiral Model. As the pre- and post-implementation diagrams are compared, not only is the question of whether defects are present to be answered, but also what degree of risk exists by continuing the development process if defects are suspected. This risk assessment is based upon comparing the diagrams, and ,if defects are present, determining their cause. If defects are arising from an incorrect or incomplete design then it is obvious that additional work must be performed to correct or expand the design before further completion of the software can occur.

Because the technique employs both static analysis of the software to produce class diagrams and

dynamic analysis to produce sequence and collaboration diagrams, the technique corresponds to the object-oriented model which is concerned with both structure and behavior. Many aspects of object-oriented software, particularly software written in C++, can not be observed from simply examining the source code. This is one of the limitations of data flow analysis represented by [HARROLD, 1989] and discussed in Section 4.3.1.

7.2.3 Criterion 3 - Integration of Technique Into The Object-Oriented Life Cycle

In Section 5.3 the relationship of integration testing to the object-oriented life cycle is discussed. The Revised Spiral Model, chosen for the software development life cycle, was modified, see Figures 5.4 and 5.5, to provide for additional insight into when and where integration testing should occur. The proposed integration testing technique is designed to merge with the Revised Spiral Model in such a way as to improve the development process as related to integration testing.

As stated in Section 2.7, software testing is only effective when it locates flaws in the software. The technique is suggested as a means of performing integration testing, even though it has been found that the technique does demonstrate usefulness in performing design verification and system testing.

The usefulness of a testing tool or technique is expanded when it one can be used for more than its primary purpose of identifying defects in the software. It is recognized that the integration testing technique is also effective in both examining the design and tracing that design to implementation. Further research into these two benefits is suggested.

7.2.4 Criterion 4 - Scalability of Technique to Larger Projects

With all proposed integration techniques it is important to evaluate their applicability to larger and

more complex systems. The limited simulation presented in Chapter 6 does not adequately represent the use of the integration testing technique in a 'real world' environment. Further research is required before one can draw a positive conclusion regarding the scalablity of the technique.

Utilizing a controlled experiment would assist in the evaluation of the effectiveness of the technique when considering its usage in larger projects. Difficulties exist in developing a repeatable experiment.

Still, from examining the steps required to implement the integration testing technique it is certain that a larger degree of automation is required before the technique can be evaluated within the context of a large scale development project. Weaknesses related to this issue of scalability are discussed further in Section 7.3.

7.3 Recognized Weaknesses

One of the weaknesses in the technique is the requirement for the various models to be created manually, along with assistance from a CASE tool, in this demonstration Rational Rose, in reverse engineering the class diagrams and a debugger for program tracing. This fact causes the expenditure of a considerable amount of time in analyzing the software and actually inserting data production statements. The manual insertion led to the potential flawing of the software. Defects introduced by manual editing can result in failure of the software to compile, adding to the time required to complete testing. No technique one uses should require the modification of the actual code since that can invalidate any previous testing effort. Because the individual methods were actually modified, both internally and at the parameter level, defects were easily introduced.

The importance of object constructors and destructors is recognized in developing object-oriented

software, particularly in C++. The language restrictions of C++ prohibit the use of parameterized destructors. This fact caused the use of global file streams in order to collect data relating to the destruction of objects. The object destruction sequence is important in order that one can see that proper garbage collection is taking place and objects are not left orphaned in memory.

Use of the *return* statement in the *main* function also caused problems when using the manual technique. Objects are destroyed after the *return* statement is encountered. Closing the data collection file prohibited the capture of the object destructor information.

Once the data were produced by the execution of the instrumented program, the data were again manually analyzed. A model was produced representing the dynamic view of the system. Comparisons were made and areas where the post-implementation model differed from the design model were noted. Here again the process was primarily manual with the potential for defects to be either introduced or overlooked.

The weaknesses have a negative impact on the usability of the technique. Similar to manual code reviews and inspections, there are considerable costs associated with using the technique in its current nonautomated state.

7.4 Recognized Strengths

The technique of performing post-implementation analysis of the software and the development of models representing the same views of the software as those used for the development was helpful in locating areas where the implementation deviated from or added to the design. Using that information, areas were highlighted that required the creation of additional test cases and/or scenarios. Expanding the number of test cases increased the likelihood of locating defects. In addition, areas where variations between the design and the post-implementation diagrams were

identified, more investigation was needed to determine the cause.

The actual testing of object-oriented software still utilizes many of the same techniques as those used in testing procedurally-based software. The differences relate to the need for both expanded testing and for the creation of test suites that exercise the software through the various objects, not just the member functions. System level testing must also recognize the object-oriented nature of the software.

7.5 Impact of Technique on Software Testing

The testing technique presented in this thesis is designed to assist the user in performing integration testing of object-oriented software. The technique provides one with a means of presenting graphical representations of the software under development. By using UML diagrams for both the analysis and design diagrams and the post-implementation representations, one has a means by which the requirements and design specifications can be compared with the resulting software. This ability to compare the desired with the actual software allows one the opportunity to identify flaws in the software. Furthermore, test cases can be developed that exercise the software in a way that corresponds with the implementation. Weaknesses in defect capture and recovery, which are more related to unit testing, can be identified by using the post-implementation views for test case generation. The technique expands the view of the software by breaking the software into components and clusters.

The use of the technique suggested in this thesis is considered beneficial in the testing of object-oriented software written in C++. Even with the limitations outlined in the previous sections, it is apparent that the use of the analysis techniques does offer important insight into the structure and behavior of the software. Using this technique integration and analysis will occur with each iteration of the development life cycle. Early and continuous analysis helps reduce late integration problems.

To properly provide for a design that can assist in the coding of the software and the testing of that software, it is necessary that one have a detailed design that contains:

> interaction diagrams with necessary parameters and values for messages,
> identification of the responsibilities of each object's procedures, and
> creation and definition of each object's data structure and algorithms.

The detailed design is used to support both the development and testing of the software.

The nature of object-oriented software requires different levels of testing, inter- and intra-method testing, class, components, and sub-system testing. Static analysis, much like a profiler, was able to identify weaknesses in the software due to its close relationship to code inspections. Areas where the software was less-than-clear were identified and improvements were made.

7.6 Future Research Efforts

As mentioned above, the techniques presented in this paper are manual. Additional work needs to be performed that would allow for the automation of the analysis of both the source and the executing versions of the software. This automation would help overcome the weaknesses in the manual process. Effort must be made to develop a program capable of processing programs written in C++ and producing the class diagrams. Further enhancements would then take the original design and compare it to this class diagram. The resulting comparison could then be displayed graphically, either on a display screen or in hard copy.

In regard to the analysis of the executing program, effort should be made to understand the program as it executes without the use of code modifying data production statements. Work should be carried out that focuses on the use of monitor software that allows the code to run in

a simulated operating system environment. As software interrupts occur, as methods are called, the monitor software could generate the dynamic and functional views. Integrating this tracing with the source code, very much like a debugger, would enhance program understanding. Source code tracing does not reveal information about the compiled code that resides in external libraries. Calls made from the source would be available, along with messages returned from the member functions.

The use of templates in C++ impacted the traceability of the source code. The actual code generated by a template does not exist until the code is compiled. This fact, along with the use of late binding and dynamic linking libraries, prohibits one from having a clear view of the software. Debuggers do allow for tracing within certain libraries.

Additional research is needed in order to perform a more thorough evaluation of the technique. Experimental options are discussed in Section 7.6.1.

7.6.1 Experimental Design

It is recognized that the scope of the demonstration of concept, as presented in this thesis, is quite limited. Expansion of the simulation would greatly benefit evaluation of the integration testing technique. Utilizing two independent groups, one applying the technique, and one using a more traditional software development/integration method would allow for a comparative analysis of the effectiveness of the technique. Experimentation within computer science, however, is quite expensive due to the considerable costs involved.

To properly evaluate the ability of a technique to identify the presence of defects, one must have control over the type and number of defects present in the system. Since the focus of this thesis is the ongoing comparison of pre- and post-implementation diagrams it is necessary that one have

some means of controlling the type of defects being introduced, as well as the time at which they are to occur. Relying upon two independent groups, one applying the technique and one using some other method, would not present a fair evaluation of the technique since either group could introduce defects at different locations within the software, and at different frequencies.

The development of such an experiment could be structured along the following lines.

Table 7.3 Design of Controlled Experiment for Integration Testing Technique

Group #1		Group #2	
Design Defective	Design Not Defective	Design Defective	Design Not Defective
Applying Other Integration Testing Technique	Applying Other Integration Testing Technique	Apply Integration Testing Technique of Thesis	Apply Integration Testing Technique of Thesis

As one can see from Table 7.3, there are different groups necessary to obtain repeatable results where the integration testing technique can have any hope of being properly evaluated. If the design is defective then the situation can arise where the programmers followed the design and as pointed out in Section 6.3.2, the diagrams representing the design and the implementation should be equivalent. Careful inspection of the diagrams, however, may reveal flaws in the design. Those defects are more likely to be found during system testing.

As for the selection of the other integration testing technique to use for the control group 1, defects or weakness in that integration testing technique may misidentify defects in the controlled project. Furthermore, as discussed in the scenarios in Section 5.6, it is possible that the programmers may introduce additional defects into the controlled system.

For these reasons, along with the aforementioned costs, experimentation in software engineering and in this particular case with the integration testing technique, is problematic. Perhaps a more realistic evaluation would be the use of a existing software system to produce the UML diagrams representing the implementation. If the system was originally developed using UML for the design then one could conceivably compare the diagrams representing the design and those created through reverse engineering. The analysis and comparison of the diagrams would then be used to search for the presence of defects remaining in the system. If defects were found then the technique would be shown to have merit. If no additional defects are found then the value of the technique would still be in question.

A third method of evaluating the system would be to have a team develop a complete system utilizing the technique. After the system is complete one or more integration/system testing techniques could be employed on the completed system. Any serious defects encountered in the post development testing and evaluation phase would certainly point to weaknesses in the integration testing technique and could raise issues concerning where improvements in the technique needs to be made. If no additional defects are found then the integration testing technique would be shown to exhibit value.

Before the technique can be used for the development of a large and complex system the automation of the post-implementation diagrams must be addressed. In addition, research in the automation of comparison of the pre- and post-implementation diagrams should be completed. Manually creating and comparing the diagrams is a major drawback in terms of cost. However, as has been shown by code reviews and walkthroughs, the manual review of the diagrams may reveal subtle defects which could possibly be overlooked by an automated tool.

7.7 Summary

From the discussion of the technique presented in Chapter 6 and the review of the technique

presented in this chapter, it is possible to see the potential benefits of using post-implementation models of the software to assist in testing. Test suites can be developed that are designed to exercise the actual code. Deviation from design models assists in both defect identification and the recognition for the need of expanded testing.

The integrated nature of object-oriented software development is exemplified by the use of UML as the modeling language and the Revised Spiral Model as the software development life cycle process. This integrated view of software development reenforces the need for an integration testing technique that provides for testing throughout the life cycle.

Weakness in the method centers around the lack of automation in the analysis and diagram creation. Additional work in the automation of tools is needed. The lack of automated analysis and comparison tools led to considerable time being spent on instrumenting the code and evaluating the resulting data. Furthermore, the process of manually examining the code and making modifications is defect prone. Automated analysis would save time and expand the application of the technique to large software projects.

Even with the known weaknesses in the technique, the use of UML to model post-implementation views of the software contributes to software understanding and testing. Continuously integrating, modeling, and testing assist in the development of accurate and quality-based software. With an understanding of the limitations of the technique, it is considered that the technique is beneficial in the analysis and testing of object-oriented software. The user is able to produce the fundamental views used in object-oriented design. Comparison of the pre- and post-implementation models of the software allows one to identify areas where the software does not conform with the design. In those areas where the design does not match the resulting implementation one should focus on how and why the two pictures of the same view are not consistent with each other. In some cases the difference may be the result of more detailed views generated from the actual software rather than from the design. However, in other instances the

differences will pinpoint defects in the implementation that would not necessarily have been illuminated by traditional testing techniques.

REFERENCES

[Alexander, 1999]. R. Alexander, "Testing the Polymorphic Relationships of Object-Oriented Components," Technical Report ISSE-TR-99-05, Department of Information and Software Engineering, George Mason University, 1999.

[Alexander and Hutchinson, 1997]. R. Alexander, J. Payne, and C. Hutchinson, "Design for Testability for Object-Oriented Software," Object Magazine, July 1997, 34++.

[Alexander and Offutt, 1999]. R. Alexander and A. Offutt, "Analysis Techniques for Testing Polymorphic Relationships," Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '99), pages 104-114, Santa Barbara California, 1999.

[Alhir, 1998]. S. Alhir, *UML In A Nutshell,* O'Reilly, Cambridge, Massachusetts, 1998.

[Ambler, 1996]. S. Ambler, "Testing Objects," Software Development, August, 1996.

[Becker, 1997]. P. Becker, "Questions and Answers," C/C++ Users Journal, July 1997, 85-88.

[Beizer, 1984]. B. Beizer, *Software System Testing and Quality Assurance*, Van Nostrand Reinhold, New York, New York, 1984.

[Beizer, 1990]. B. Beizer, *Software Testing Techniques (2nd Edition)*. Van Nostrand Reinhold, New York, New York, 1990.

[Bellcore, 1993]. "TR-NWT-000179, Reliability and Quality Generic Requirements (RQGR), Quality System Generic Requirements for Software," Bellcore, Issue 2, June 1993.

[Berard, 1996]. E. V. Berard, "Issues in the Testing of Object-Oriented Software," (1996). http://www.toa.com/pub/.../oo-testing_article.text (26 Sept 1996).

[Berard, 1996a]. E. V. Berard, "Be Careful with "Use Cases"", (1996). http://www.toa.com/pub/html/use_case.html (26 Sept 1996).

[Bergman-Terrell, 1991]. E. Bergman-Terrell, "A Call-Tree Generator for C Programs," C/C++ User's Journal, October, 1991.

[Biggs, 1996]. P. Biggs, "A Survey of Object-Oriented Methods," University of Durham, UK (1996). http://www.dur.ac.uk/~dcs3pjb/survey.html (6 Dec 1996).

[Binder, 1995]. R. V. Binder, "Trends in Testing Object-Oriented Software, " Computer, vol. 28, no. 10, October, 1995, 68-69.

[Binder, 1995a]. R. V. Binder, "Object-Oriented Testing: Myth and Reality, " Object Magazine, 1995.

[Binder, 1995b]. R. V. Binder, "The FREE Approach to Testing Object-Oriented Software: An Overview," (1995). http://www.rbsc.com/pages/FREE.html#"SUM". (14 March 1996).

[Binder, 1995c]. Binder, R. V., "State-based testing," Object Magazine. Vol. 5, No. 4, 75-78, 1995.

[Binder, 1996]. R. V. Binder, "The FREE Approach for Software Testing: Use Cases, Threads, and Relations," Object Magazine, vol. 6, no. 2, February 1996.

[Binder, 1996a]. R. V. Binder, "Summertime and the Testin' is Easy...", Object Magazine, November, 1996, 72++.

[Binder, 1997]. R. V. Binder, "Object-Oriented Testing: What's New?," Object Magazine, July 1997, 21++.

[Boehm, 1988]. B. W. Boehm, "A Spiral Model of Software Development and Enhancement," Computer, May 1988, 61-72.

[Boggs and Boggs, 1999]. W. Boggs and M. Boggs, *Mastering UML with Rational Rose,* SYBEX, San Francisco, California, 1999.

[Boisvert, 1997]. J. Boisvert, "OO Testing in the Ericsson Pilot Project," Object Magazine, July, 1997, 26+.

[Booch, 1994]. G. Booch, *Object-Oriented Analysis and Design With Application, Second Edition.* Addison-Wesley, Redwood City, California, 1994.

[Booch, 1996]. G. Booch, *Object Solutions: Managing the Object-Oriented Project.* Addison-Wesley Publishing Company Inc., Menlo Park, California, 1996.

[Booch, 1996a]. G. Booch, "The Unified Modeling Language," Unix Review, December 1996.

[Booch, et al, 1999]. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language Users Guide,* Addison Wesley, Reading, Massachusetts, 1999.

[Card, 1990]. D. Card, "Software quality engineering, " Information and Technology, vol. 32, no. 1, January/February 1990.

[Carlson, 1997]. P. Carlson, "An Automated Testing Tool for Win16," C/C++ Users Journal,. July 1997, 25-37.

[Chaudhry, 1997]. P. Chaudhry, "A New Trace Class," C/C++ Users Journal, June 1997: 51-52.

[Coallier, 1995]. F. Coallier, "TRILLIUM: A Model for the Assessment of Telecom Product Development & Support Capability," Software Process Newsletter, Winter 1995, 3+.

[Corriveau, 1996]. J. Corriveau, "Traceability Process for Large OO Projects," Computer, September 1996.

[Crosby, 1979]. P. B. Crosby, *Quality Is Free*. The New American Library, Inc., New York, New York, 1979.

[D'Souza and Wills, 1999]. D. D'Souza and A. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

[de Champeaux, et. al., 1993]. D. de Champeaux, D. Lea, and P. Faure, *Object-Oriented System Development*. Addison-Wesley Publishing, Company, Reading, Massachusetts, 1993.

[Deutch, M. and Willis, R., 1988]. M. Deutch and R. Willis, *Software Quality Engineering: A Total Technical and Management Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Doong and Frankl, 1994]. R. Doong and P. Frank, "The ASTOOT Approach to Testing Object-Oriented Programs," ACM Transactions on Software Engineering and Methodology, vol. 3, no. 4, April 1994, 101-130.

[Duncan, et al., 1996]. I. Duncan, M. Munro, and D. Robson, "MATOOS: The Maintenance and Testing of Object Oriented Systems," Centre for Software Maintenance, Dept. of Computer Science, Univ. of Durham. (1996). http://www.ac.uk/~d...ects/matoos/matoos.html. (3 Oct 1996).

[Dunn, 1990]. R. Dunn, *Software Quality: Concepts and Plans*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1990.

[du Plessis, 1990]. A. L. du Plessis, OOISEE Project Proposal, Technical Report TR 90/1, Department of Computer Science and Information Systems, University of South Africa, 1990.

[du Plessis and van der Walt, 1992]. A. L. du Plessis and E. van der Walt, "Modeling the

software development process," Poster Session at IFIP WG8.1 Working Conference - ISCO2, Alexandria, Egypt, April 1992.

[Emam, et al., 1997]. K. Emam, J. Drouin, and W. Melo. *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination.*, IEEE Computer Society, Los Alamitos, California, 1997.

[Ericksson and Penker, 1998]. H. Ericksson and M. Penker. *UML Toolkit.* John Wiley & Sons, Inc., New York, New York, 1998.

[Firesmith, 1996]. D. Firesmith, "Pattern language for testing object-oriented software," Object Magazine, Jan 1996, 32-38.

[Fowler and Scott, 1997]. M. Fowler, and K. Scott, *UML Distilled.* Addison-Wesley, Reading, Mass., 1997.

[Frank, et. al., 1996]. B. Frank, P. Marriott, and C. Warzusen, *The Software Quality Engineering Primer.* Quality Council of Indiana, West Terre Haute, Indiana, 1996.

[Glass, 1992]. R. Glass. *Building Quality Software.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[Harmon and Watson, 1998]. P. Harmon and M. Watson, *Understanding UML: The Developer's Guide.* Morgan Kaufmann Publishing, Inc., San Francisco, CA, 1998.

[Harrold and Soffa, 1991]. M. J. Harrold and M. L. Soffa, "Selecting and Using Data for Integration Testing," IEEE Software, March, 1991, 58-65.

[Harrold, 1994]. M. J. Harrold, "Incremental Testing of Class Structures," Department of Computer Science, Clemson University, Department of Computer Science, Clemson, South Carolina, 1994.

[Harrold and Rothermel, 1995]. M. J. Harrold and G. Rothermel, "Structural Testing of Object-Oriented Classes, " Proceedings 8th Annual Software Quality Week, May 1995, Software Research, Inc, San Francisco, CA.

[Henricson and Nyquist, 1992]. M. Henricson and E. Nyquist, "Programming in C++, Rules and Recommendations." (1992). http://www.rhi.is/~h...rules.html#introduction. (6 Dec 1996).

[Hetzel, 1988]. B. Hetzel, *The Complete Guild to Software Testing*, Second Edition, QED Information Sciences, Inc. Wellesley, Massachusetts, 1988.

[Humphrey, 1989]. W. Humphrey, *Managing the Software Process,* Addison-Wesley, Reading , Massachusetts, 1989.

[Humphrey, 1995]. W. Humphrey, *A Discipline for Software Engineering,* Addison Wesley, Reading, Massachusetts, 1995.

[Hunt, 1996]. N. Hunt, "Performance testing C++ code," Journal of Object-Oriented Programming, January, 1996, 22-25.

[Hymowech, 1988]. Hymowech, "Find That Function!," Dr. Dobbs Journal, August 1988, 70+.

[IEEE, 1987]. IEEE, *ANSI Standard: IEEE Guide to Software Configuration Management,* ANSI/IEEE Std. 1042-1987, IEEE, New York, New York, 1987.

[IEEE, 1990] IEEE, *Glossary of Software Engineering,* IEEE, New York, New York, 1990.

[Interactive, 1998]. Interactive Software Engineering, "Eiffel in a Nutshell,", (1998). http://www.eiffel.com/eiffel/nutshell.html.

[Jacobson, et al., 1992]. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Reading, Massachusetts, 1992.

[Jacobson, et al., 1999]. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process,* Addison Wesley, Reading, Massachusetts, 1999.

[Jorgensen, 1995]. P. Joregensen, *Software Testing: A Craftsman's Approach.* CRC Press, Boca Raton, Florida, 1995.

[Jorgensen and Erickson, 1994]. P. Jorgensen and C. Erickson, "Object-Oriented Integration Testing," Communications of the ACM, vol. 37, no. 9, September 1994, 30-38.

[Kuvaja, et al., 1994]. P. Kuvaja, J. Simila, L. Krzanik, A. Bicego, S. Saukkonen, G. Koch, *Software Process Assessment and Improvement: The BOOTSTRAP Approach,* Blackwell Publishers, 1994.

[Kerningham and Pike, 1999]. B. Kerningham and R. Pike, *The Practice of Programming,* Addison Wesley, Reading, Massachusetts, 1999.

[Konrad and Paulk, 1995]. M. Konrad, M. Paulk, and A. Graydon, "An Overview of SPICE's Model for Process Management," Proceedings of the Fifth International Conference on

Software Quality, Oct 1995.

[Kung, 1995]. D. Kung, "Developing an OO Software Testing and Maintenance Environment," Communications of the ACM, October, 1995, 75-87.

[Layman and Layman, 1997]. B. Layman and J. Layman, "Road to Qualityville," Windows Tech Journal, August 1997, vol. 6, no. 8, 18+.

[Lee and Tepfenhart, 1997]. R. Lee and W. Tepfenhart, *UML and C++, A Practical Guide to Object-Oriented Development*. Prentice Hall, Upper Saddle River, New Jersey, 1997.

[Liu, 1996]. W. Liu, "Selecting Test Cases for Object Oriented Programs". (1996). http://www.wlsi.uwaterloo.ca/~wbliu/proposal.html. (11 Mar 1997).

[Manning, 1997]. M. Manning, "What to Wear," Windows Tech Journal, August 1997, 39+.

[Marick, 1995]. B. Marick, *The Craft of Software Testing*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1995.

[Marick, 1996]. B. Marick, "Notes on Object-Oriented Testing Part 1: Fault-Based Test Design," (1996]. http://www.stlabs.com/marick/1-fault.html (24 Sept 1996).

[Martin and Odell, 11998]. J. Martin and J. Odell, *Object-Oriented Methods: A Foundation*. Prentice Hall PTR, Upper Saddle River, New Jersey, 1998.

[McConnell, 1993]. S. McConnell, *Code Complete*. Microsoft Press, Austin, Texas, 1995.

[McGregor and Korson, 1994]. J. D. McGregor and T. Korson, "Integrating Object-Oriented Testing and Development Processes, " Communications of The ACM, vol. 37, no. 9, September 1994, 59-77.

[Meyer, 1992]. B. Meyer, *Eiffel: The Language*, Prentice Hall, Upper Saddle River, New Jersey 1993.

[Meyer, 1997]. B. Meyer, *Object-Oriented Software Construction, 2nd Ed.,* Prentice Hall, Upper Saddle River, New Jersey, 1997.

[Myers, 1979]. G. J. Meyers, *The Art of Software Testing*. John Wiley and Sons, New York, New York, 1979.

[Nutter, 1988]. S. Nutter, "An Aid To Documenting C," Dr. Dobbs Journal, August 1988, 40+.

[Paulk et al., 1995]. M. C. Paulk, C. V. Weber, B. Curtis, M. B. Chrissis, *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, Reading, Massachusetts, 1994.

[Payne, 1997]. J. E. Payne, "Testing Web Server Content," Software QA, vol. 4, no. 3, 1997, 6-7.

[Perry, 1991]. W. Perry, *Quality Assurance for Information Systems: Methods, Tools, and Techniques*. QED Technical Publishing Group, Boston, Massachusetts, 1991.

[Perry and Kaiser, 1990], D. E. Perry and G. E. Kaiser, "Adequate Testing and Object-Oriented Programming," Journal of Object-Oriented Programming, vol. 2, no. 5, January/February, 1990, 13-19.

[Pinheiro and Goguen, 1996]. F. Pinheiro and Goguen, J., "An Object-Oriented Tool for Tracing Requirements," IEEE Software, March 1996.

[Poston, 1994]. R. Poston, "Automated Testing from Object Models," Communications of the ACM, vol. 37, no. 9, September 1994, 48-58.

[Pressman, 1997]. R. Pressman, *Software Engineering: A Practitioner's Approach, 4th Ed.*. McGraw-Hill Companies, Inc., New York, New York, 1997.

[Quatrani, T., 1998] Quatrani, T., *Visual Modeling with Rational Rose and UML*. Addison-Wesley, Reading, Massachusetts, 1998.

[Rational, 1997]. *UML Summary, Version 1.1*. Rational Software Corporation, Santa Clara, CA, 1 September 1997.

[Rational, 1997a]. *UML Notation Guide, Version 1.0*. Rational Software Corporation, Santa Clara, California, 1997.

[Rational, 1997b]. *UML Process Specification Extensions, Version 1.1*. Rational Software Corporation, Santa Clara, California, 1 September 1997.

[Rational, 1997c]. *UML Semantics, Version 1.1*. Rational Software Corporation, Santa Clara, California, 1 September 1997.

[Rational, 1997d]. *UML Extensions of Objectory Process for Software Engineering, Version 1.1*. Rational Software Corporation, Santa Clara, California, 1 September 1997.

[Rational, 1997e]. *UML Semantics Appendix M1-UML Glossary*. Rational Software

Corporation, Santa Clara, California, 13 January 1997.

[Rational, 1997f]. "Rational Objectory Process 4.1 - Your UML Process", Whitepaper, Rational Software Corporation, Santa Clara, California, 1997.

[Rational, 1997g]. "Reaching CMM Levels 2 and 3 with the Rational Objectory Process", Rational Software Corporation, Santa Clara, California, 1997.

[Royce, 1970]. W. Royce, "Managing the Development of Large Software Systems: Concepts and Technique, " 1970 WESCON Technical Papers, Western Electronic Show and Convention, Los Angeles, August 1970, pp. A/1-1-A/1-9. Reprinted in: Proceedings of the 11th International Conference on Software Engineering, Pittsburgh, may 1989, pp. 328-38.

[Rumbaugh, et al., 1991]. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Rumbaugh, 1996]. J. Rumbaugh, *OMT Insights*. SIGS Books, New York, New York, 1996.

[Rumbaugh, et al., 1999]. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual,* Addison Wesley, Reading, Massachusetts, 1999.

[Sanders and Curran, 1994]. J. Sanders and E. Curran, *Software Quality: A Framework for Success in Software Development and Support*. Addison-Wesley Publishing Company, Workingham, England, 1994.

[Schach, 1999]. R. Schach, *Classical and Object-Oriented Software Engineering, 4th Ed.* McGraw-Hill , Boston, Massachusetts, 1999 .

[Sommerville, 1996]. I. Sommerville, *Software Engineering, 5th Ed.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[Steenkamp, 1996]. L. Steenkamp, Communications between G. W. Skelton and L. Steenkamp, October, 1996, March, 1997.

[Stroustrup, 1991]. B. Stroustrup, *The C++ Programming Language, Second Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[Tai and Daniels, 1999]. K. Tai and F. Daniels, "Interclass Test Order for Object-Oriented Software," Journal of Object-Oriented Programming, July/August, 1999, 18+.

[Texel and Williams, 1997]. P. Texel and C. Williams, *Use Cases Combined with Booch, OMT, UML.* Prentice-Hall, Upper Saddle River, New Jersey, 1997.

[Thornton, 1999]. D. Thornton, *A Quality Assurance Reference Model for Object-Orientation*, Masters Dissertation, University of South Africa, June 1994.
[Thornton and du Plessis, 1995]. D. Thornton and A. L. du Plessis, "A Quality Assurance Reference Model for Object-orientation," *Proceedings of the SAICSIT-95 Conference*, Pretoria, South Africa, May 1995.

[Turner and Robson, 1992]. C. D. Turner and D. J. Robson, "A Suite of Tools for the State-Based Testing of Object-Oriented Programs, Technical Report: TR-14/92," Durham, England: Computer Science Division, School of Engineering and Computer Science (SECS),University of Durham, 1992.

[Turner and Robson, 1993]. C.D. Turner and D. J. Robson, "The State-based Testing of Object-Oriented Programs," IEEE, Proceedings of the Conference on Software Maintenance, 1993.

[Turner and Robson, 1994]. C. D. Turner and D. J. Robson, "The Testing of Object-Oriented Software, Technical Report: TR-13/92," Durham, England: Computer Science Division, School of Engineering and Computer Science (SECS), University of Durham, 15 January 1994.

[van der Walt and du Plessis, 1994]. E. van der Walt and A. L. du Plessis, "A Revised Spiral Model for Object-Oriented Development," Proceedings of the Fourth International Conference on Information Systems Development, ISD'94, Slovenia, 1994.

[Walton, 1986]. M. Walton, *The Deming Management Method*. The Putnam Publishing Group, New York, New York, 1986.

[Weinberg, 1992]. G. Weinberg, *Quality Software Management, Vol. 1: Systems Thinking*. Dorset House Publishing, New York, New York, 1992.

[Wirfs-Brock et al., 1990]. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

APPENDIX

APPENDIX A

Unified Modeling Language Overview

CONTENTS

A.1 Introduction

Appendix A provides an overview of the composition of the Unified Modeling Language (UML) by illustrating the various diagrams available and the use of each in presenting a model of the different views of a software system. This appendix is not to be considered an extensive tutorial or a complete reference for UML but, rather, is designed to provide additional information that may assist one in understanding the material presented in this thesis.

UML was developed by Booch, Rumbaugh, and Jacobson in an attempt to develop a standardized language for modeling object-oriented software that is process independent. The developer is allowed to chose a software process model best suited for a particular domain of discourse. UML

was designed to assist the user in building models, e.g. diagrams, of the software under development. These diagrams are used to record the analysis and design specifications. The models can be classified into the following categories:

1) logical design models, and
2) physical architecture models.

The models can represent either a static or dynamic view of the system under design.

A.2 Logical View of Software

The logical view of an object-oriented software systems represents the external interface and use of the system from the user's perspective. UML provides two different types of diagrams to represent this external view, Use Case diagrams and Class diagrams. Use case models provide one with representations of the users and the system. Actors are presented and the relationship/interaction with other actors and with the system are illustrated.

A.2.1 Use Case Diagrams

Use case diagrams are used to show the interaction between the system and individual actors. The system is treated as a black-box and the-use case illustrates a specific use of the system. The meta-primitives of a use case diagram are

Functional requirements of a system are defined through use cases.

A.2.2 Class Diagrams

Classes, objects, and their relationships are illustrated through the use of class and diagrams and relationship diagrams. Class diagrams give a static view of the software. They can also illustrate the association and relationship among the classes. Class diagrams can show association, recursive association, qualified association, or association, ordered association, ternary association, aggregation, generalization, dependency, and refinement. These diagrams provide the user with information about the class structure of the software.

Class diagrams are used to illustrate the classes of which the system is composed. Class diagrams are a static modeling type in that they illustrate classes and their relationship to other classes without information about the dynamic interaction of objects. Class diagrams, depending upon their level of detail, can show only the names of classes, the class name and attributes, or the complete class with its name, attributes, and methods. The meta-primitives of the class diagram are as follows:

- class name,
- attributes,
- methods, and
- directed arrows.

In addition to these primitives, the class diagram also shows the relationship of one class to another by the use of arrows. These relationships are:

- normal association,
- generalization,
- dependency, and

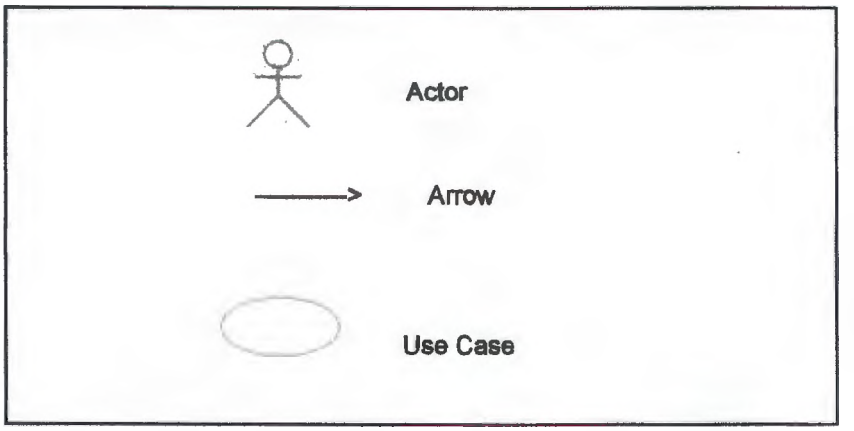


Figure A1 Use Case Primitives

Use case diagrams are created early in the analysis/design of the system. Specific implementation is not considered at the time these diagrams are drawn. Still, use case diagrams are helpful in determining the use of a system. They illustrate various relationships such as generalization, association, dependency between the elements of use case diagrams: the actors, the system, and the use cases. The following diagram illustrates the generic concepts of the use case diagram.

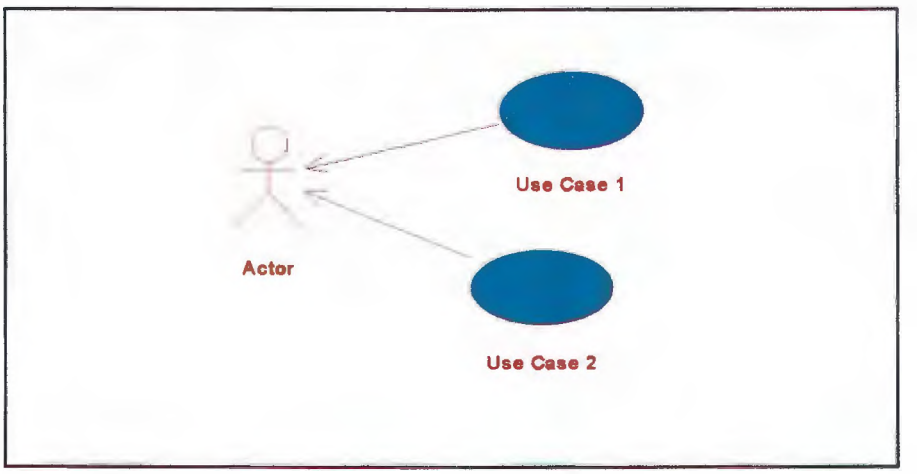


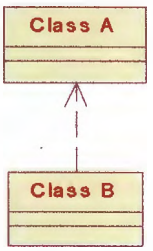
Figure A2 Use Case Diagram

- refinement.

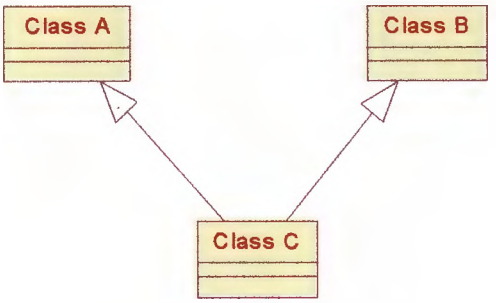
These basic relationships are shown as:

Class Relationships

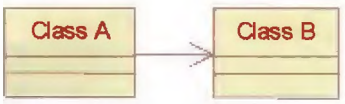
- dependency



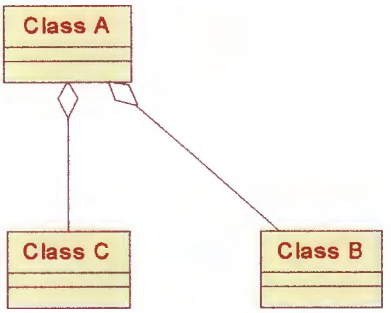
- generalization



- association



- aggregation



These class diagrams illustrate different associations which are possible among classes and objects. Using these diagrams one is able to create class/object models that represent the desired relationship within the software.

A.3 Physical View of Software

The physical view of software represents the internal construction. Here the software can be

viewed in either a static or dynamic model. The static models reflect structure. Dynamic models reflect the behavior of the software. In UML there are three primary dynamic models, 1) state, 2) sequence, and 3) collaboration diagrams. Each of these models provides a different view of the software's behavior.

Arrows and lines are used in each of the models to show association and to illustrate the different types of communication objects. For communication purposes the arrows indicate the direction and type of the communication. The types of communication are listed below:

- synchronous,
- asynchronous,
- simple, and
- synchronous with immediate return.

Synchronous communication occurs when Object A passes a message to Object B and then waits until a reply message is returned from Object B. Asynchronous communication is illustrated by Object A passing a message to Object B and then continuing to work without waiting for a return message from Object B indicating that its task is completed. Simple communication is used when there is a flow of control and Object A communicates with Object B; however, in the diagram no specific detail is included that defines the type of communication. Synchronous communication with an immediate return indicates that a message was sent from Object A to Object B but there was an almost immediate reply, without any substantive delay in sending that reply.

By using these various models of dynamic behavior one is able to examine the software and determine whether it is behaving correctly and consistently.

- State diagrams - show which states objects can have, the object's behavior while in that state, and which events cause an object's state to change.

- Sequence diagrams - show how objects interact and the sequence of messages that are used to perform a function; the emphasis being object interaction over time.

- Collaboration diagrams - again object interaction is key, however, the emphasis is on space, e.g. the relationship of objects while performing a particular task.

- Activity diagrams - attention is on the work being performed; here the activities are within a particular object and the order in which these activities are completed is illustrated.

The meta-primitives for dynamic models as expressed by state-based notations are:

- directed arrows indicating different types of communication and
- states.

In addition to these primitives, events are added to the diagrams. These events are the actions that occur causing an object to change from one state to another. Events are indicated on state diagrams through the use of text above/below the arrows indicating an action is occurring.

In UML there are four types of events:

- a condition becoming true,
- receipt of an explicit signal from another object,
- receipt of a call on an operation from another object, and
- passage of a designated period of time.

A.3.1 State Diagrams

A state diagram is used to illustrate the states that a particular object can have. The events that create objects, cause an object to change state, and destroy objects are included in state diagrams. The meta-primitives for the state diagram include:

- object state,
- arrows,
- events,
- conditions,
- actions, and
- responses.

In modeling object-oriented systems it is possible to have more than one state diagram. Messages can be passed between these diagrams. Essentially one models these individual diagrams as sub-states. These sub-states are then contained in blocks which can receive messages from outside of the block and issue replies depending on the change occurring within the block. In a number of instances the sub-module will not be visible unless a certain state is achieved by the other module or object. For example, a stereo component system may have a CD-player, a tape player, and a radio. Only when the stereo system is in a particular state does the corresponding object, e.g. CD-player become active, e.g. constructed. Before that, the methods, e.g. buttons, have no meaning; they don't work.

A simple state diagram can illustrate the creation of an object, an event that changes it from state A to state B and then the destruction of that object. Figure 5.5 Illustrates that state diagram.

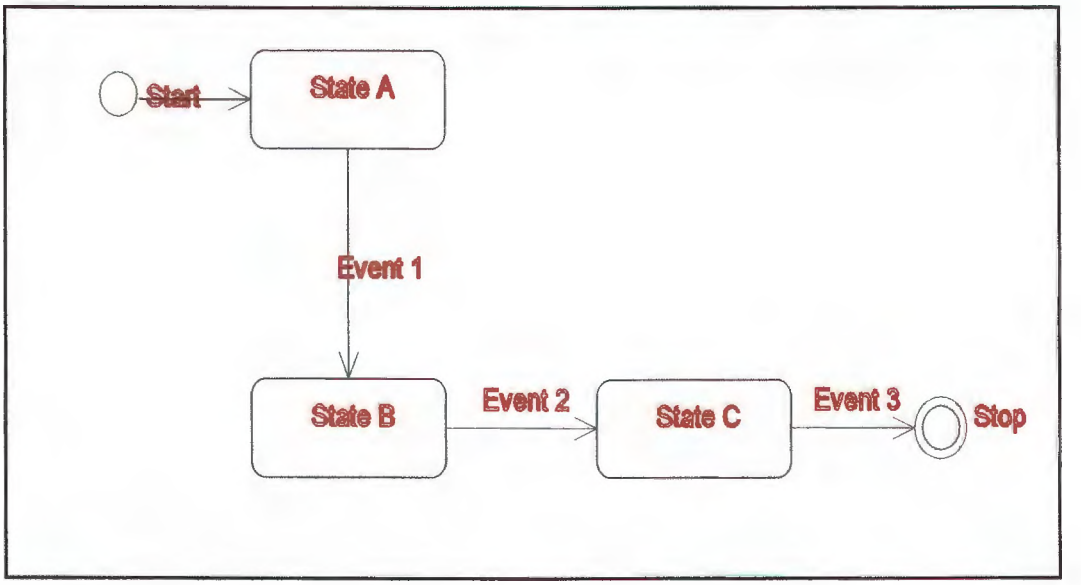


Figure A3 State Diagram

A.3.2 Sequence Diagrams

A sequence diagram targets the interaction between objects. The focus is on the message sequences, the sending and receiving of messages between objects. Sequence diagrams are concerned about a particular scenario in a given period of time. Sequence diagrams use the same primitives to draw models, as do state diagrams. However, sequence diagrams use two points of attention - the objects and time. Time is indicated on the X axis and objects on the Y axis.

In drawing sequence diagrams one uses rectangles to represent the objects. A vertical, dashed line is used to indicate the object's execution during the sequence. Horizontal arrows are used to illustrate communication between objects.

There are two forms of sequence diagrams - 1) generic and 2) instance. Instance diagrams illustrate a specific scenario in detail. Generic diagrams show all possibilities within a scenario. Therefore, the generic form has a number of different branches. To illustrate all of the instances

of scenarios, a number of different diagrams have to be drawn.

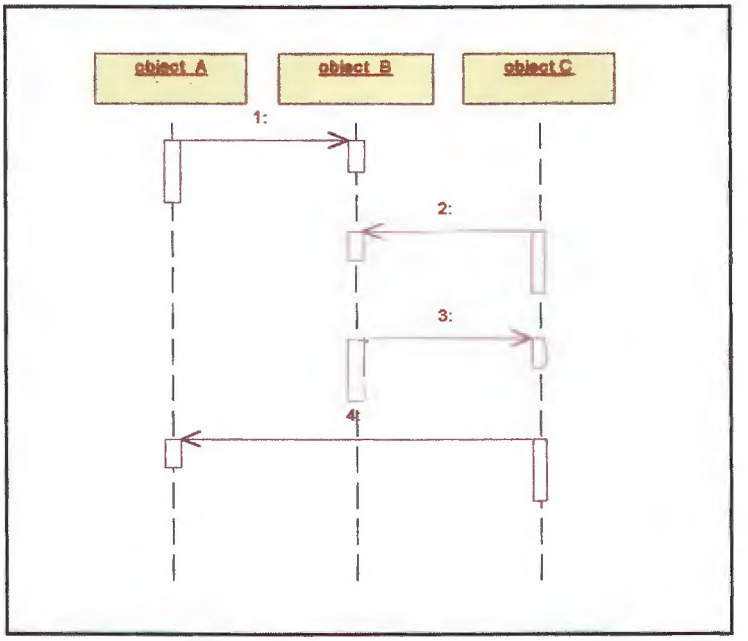


Figure A4 Sequence Diagram

This diagram illustrates the sequence of events that include intercommunication among Class A, Class B, and Class C. Class A communicates with Class B which in turn communicates with Class C. Class C returns a message to Class B which, in completion of the sequence, communicates with Class A.

Sequence diagrams provide the user with information about the relationship and communication between classes. One is able to determine both the message and the type of communication, e.g. one-way, two-way, etc, This information is vital to the development of the software.

A.3.3 Collaboration Diagrams

Collaboration diagrams, much like sequence diagrams, focus on the interaction between objects.

The collaboration diagram, instead of indicating time sequences, focuses on space. Again, these diagrams are used to model scenarios or use case execution.

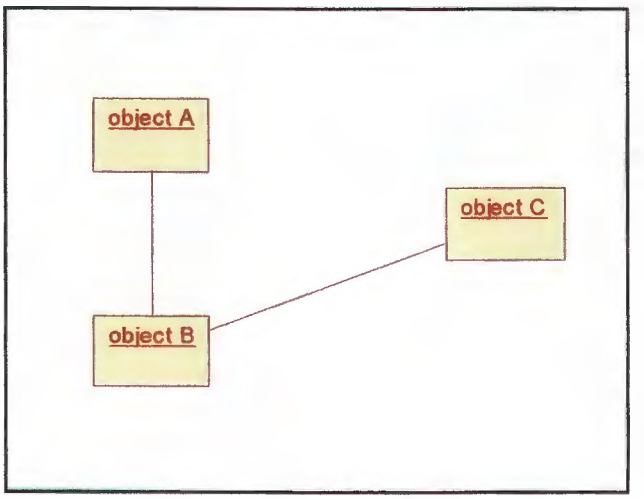


Figure A5 Collaboration Diagram

Adornments can be used to provide detail in collaboration diagrams.

Collaboration diagrams show how objects and their linkage to other objects is carried out via the ending and receiving of messages. One is concerned with which objects are collaborating and the nature of the communication between the objects.

The meta-primitives are again similar to state diagrams. The objects are drawn the same as classes in the class diagrams. Arrows indicate the direction of the communication and its nature, and message labels or adornments are provided to indicate the type of message. The meta-primitives are:

- class/object,
- message arrow, and
- message label, or adornment.

A.3.4 Activity Diagrams

This diagram is a special case of a state diagram. It is associated with an object or a class. Activity diagrams are concerned with actions and their results. Their focus is on the task being carried out within an operation, or method. Activity diagrams illustrate the internal operation of an object. Specifically they are used to do the following:

- to capture work within a specific method,
- to capture internal work of an object,
- to show how specific actions may be performed and how those actions affect surrounding objects, and
- to show a specific instance of a use case and how it will be performed in terms of operations and object state changes.

The modeling primitives are similar to those used in the other diagrams. These primitives are:

- directed arrows, indicating direction of flow of control and communication,
- rectangles for indicating methods,
- vertical lines to indicate swim lanes to separate different objects and different actions, and
- signals.

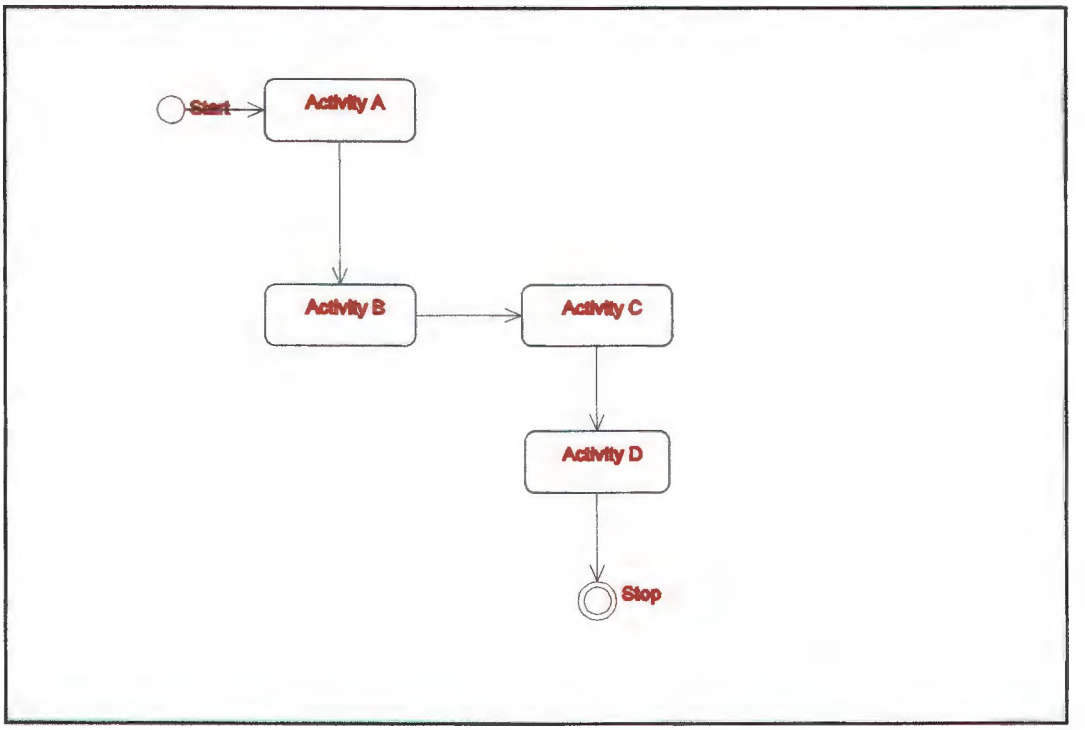


Figure A6 Activity Diagram

A.4 Component Diagrams

Component diagrams are used to model the physical attributes of a system. They describe the software components and their dependencies. The component diagram illustrates the structure of the software, including such elements as source code, object models, dynamic linked libraries, and executable modules.

The meta-primitives for the component diagrams are:

- arrows,
- component elements,
- adornments,
- sub-modules, and

- main driver modules.

Using these meta-primitives it is possible to develop a model of the entire software system. The following figure illustrates the component diagram. One can see the relationship of the main, driver and modules to the other software components.

One is able to model the different components, e.g. units, clusters, that make up the software. Depending upon one's definition of component, as referenced in 'component' diagram, it is possible to develop a model that represents that particular viewpoint.

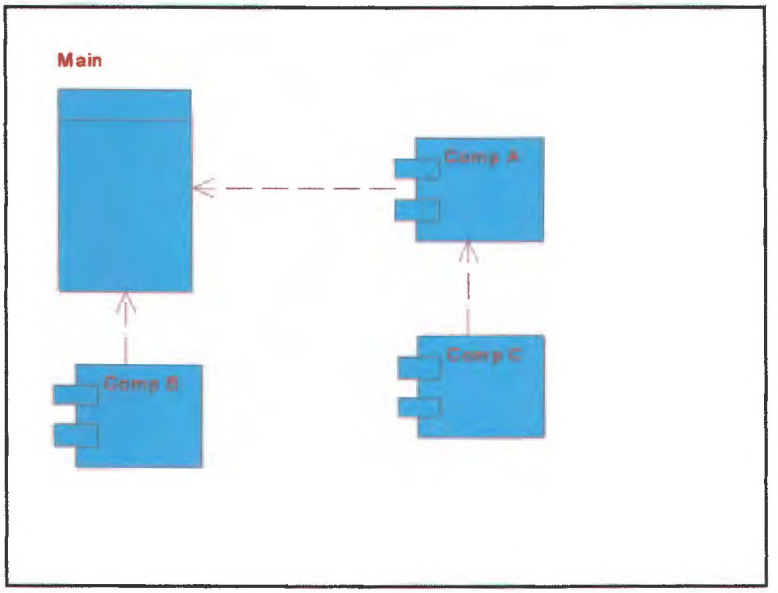


Figure A7 Component Diagram

A.5 Package Diagrams

In UML there exists the concept of the package. A package resembles a functional sub-program where closely related classes can belong to a common package. The package, or sub-program, can then communicate with other packages. Packages are used to logically organize the software. Component diagrams are similar to package diagrams except that component diagrams show the actual, physical modularization of the software. It is possible to have a one-to-one relationship of a package diagram to a component diagram.

Package diagrams use the following primitives:

- packages,
- arrows, and
- adornments.

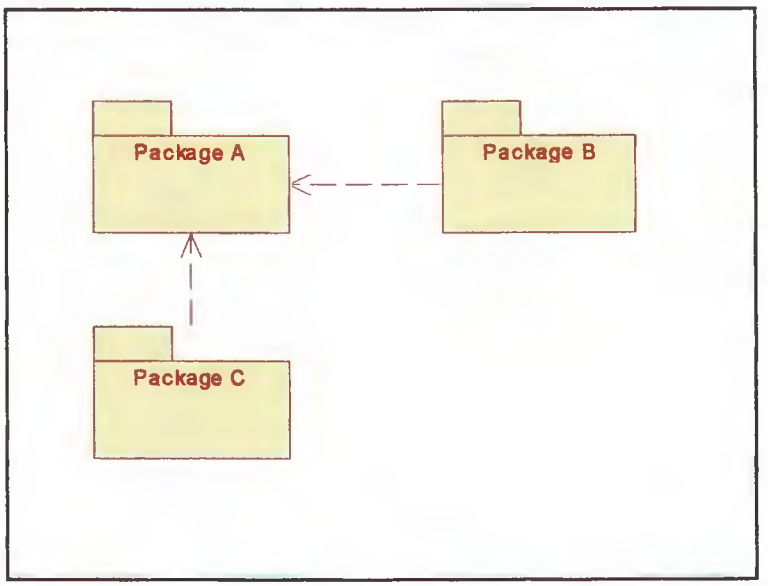


Figure A8 Package Diagram

Packages can use arrows similar to class diagrams to show association and dependency. One of the utilities of package diagrams is that they substitute for large class diagrams that do not adequately fit on the limited screen real estate. It is, therefore, a useful process to substitute class diagrams with package diagrams in large system design.