1999

Enhancing the Process of Testing Object -Oriented Systems.

Ihssan Salim Alkadi

Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

---

Recommended Citation

Alkadi, Ihssan Salim, "Enhancing the Process of Testing Object -Oriented Systems." (1999). *LSU Historical Dissertations and Theses*. 6971.

https://digitalcommons.lsu.edu/gradschool_disstheses/6971

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

**ENHANCING THE
PROCESS OF TESTING
OBJECT-ORIENTED SYSTEMS**

A Dissertation

**Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirement for the degree of
Doctor of Philosophy**

in

The Department of Computer Science

by

Ihssan Salim Alkadi

B.S., Southeastern Louisiana University, 1985

M.S., Louisiana State University, 1992

August 1999

UMI Number: 9945698

UMI Microform 9945698
Copyright 1999, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

ACKNOWLEDGMENTS

I would like to extend my sincere thanks to my advisor Dr. Doris I. Carver for her continued support and supervision through out the entire research process. I also would like to thank my wife, my twin brother, and my family for their encouragement and patience during the down times that I had while finishing this work. A special thanks goes to my father-in-law Eddie for his continued support to me and his undivided attention towards my progress of my Ph.D.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	ii
LIST OF TABLES	v
LIST OF FIGURES	vi
ABSTRACT	viii
CHAPTER 1. INTRODUCTION	1
1.1 Testing Methodologies.....	2
1.2 Automated Testing Tools.....	9
1.3 Testing for Object Oriented Programming	10
1.4 Problem Definition.....	14
1.5 Overview of research	17
CHAPTER 2. RELATED WORK.....	18
2.1 Single testing strategies.....	18
2.2 Testing frameworks with theories.....	29
2.3 Implementations of testing strategies.....	37
2.4 Assessments of testing strategies	40
CHAPTER 3. FUNDAMENTAL COMPONENTS OF AN OBJECT ORIENTED TESTING ASSISTANT	45
3.1 Object Level Test (OLT)	46
3.1.1 Object Test.....	47
3.1.2 Interface Test	49
3.2 Inheritance test	50
3.2.1 Inheritance Test Within Classes.....	50
3.2.2 Inheritance Test Complexity Estimator	51
CHAPTER 4. FUNCTIONALITY OF THE ASSISTANT.....	56
4.1 Overview.....	56
4.2 Definition of the testing assistant.....	56
4.3 Components of the testing assistant.....	57
4.4 Production of the testing assistant	58
4.4.1 OLT & TIC Test Driver Computation Algorithm	58
4.4.2 System Computation Algorithm	60
4.4.3 OLT & TIC Test Drivers Generation Algorithm	62

CHAPTER 5. ARCHITECTURE OF THE TESTING ASSISTANT	74
5.1 Overview.....	74
5.2 Overview of the System.....	74
5.3 OOTA Implementation	74
5.4 OOTA Test Data Set.....	75
5.5 OOTA Internal Structure	77
5.6 Features of OOTA	86
 CHAPTER 6. TESTING ASSISTANT CASE STUDIES	 88
6.1 OLT (Object Level Testing).	88
6.1.1 Object Testing.....	88
6.1.2 Interface Test (IT).....	91
6.2 Inheritance Test (TIC).....	96
 CHAPTER 7. SUMMARY.....	 109
7.1 Conclusions.....	111
7.2 Future Work.....	114
 REFERENCES.....	 115
 VITA.....	 120

LIST OF TABLES

2.1 Efficiency Comparison of Models.....43

5.1 OOTA Internal Structure 78

5.2 OOTA Classes, Objects, and Purpose.....82

LIST OF FIGURES

Figure 3.1	Unit Testing Example.....	48
Figure 3.2	Interface Testing Example.....	50
Figure 3.3	Single & Multilevel Inheritance Example.....	53
Figure 3.4	Multiple Inheritance Example.....	54
Figure 4.1	Test Plan Definition.....	56
Figure 4.2	Framework for the assistant.....	57
Figure 5.1	OTA Structural Layout	76
Figure 5.2	Main Menu Screen.....	76
Figure 5.3	Object Test Main Menu Screen... ..	77
Figure 5.4	Inheritance Test Main Menu Screen	79
Figure 5.5	Single Inheritance Test Menu Input Screen.....	80
Figure 5.6	Multiple/Multilevel Inheritance Menu Input Screen	80
Figure 5.7	Testprog.cpp test driver – Selective Object Test	83
Figure 5.8	person.h Declaration file for class person.....	84
Figure 5.9	Testprog.cpp test driver – Automatic Object Test	84
Figure 6.1	elevator.h Declaration file for class elevator.....	90
Figure 6.2	Declaration file billingitem.h, servie, and product	94
Figure 6.3	install.h Declaration file for class install... ..	95
Figure 6.4	Testprog.cpp test driver - interface test	96

Figure 6.5	Declaration file for employee class.....	97
Figure 6.6	Salaried, Hourly, Temporary classes	100
Figure 6.7	Faculty4.h Declaration file for class faculty.....	101
Figure 6.8	Testprog.cpp test driver - Single Inheritance	102
Figure 6.9	Declaration file for person class.....	105
Figure 6.10	Declaration file for worker class.....	106
Figure 6.11	Declaration file for manager class	107
Figure 6.12	Testprog.cpp test driver - multiple inheritance.....	108

ABSTRACT

Testing is a crucial step in the overall system development process. Using testing techniques that support features of the underlying software paradigm more effectively tests program than do testing techniques that support features of other paradigms. Systems developed with the object-oriented paradigm require techniques that support object-oriented features such as inheritance, data abstraction, encapsulation, and dynamic binding. Many techniques that are used to test systems developed with the structured paradigm are not sufficient for the testing of object-oriented systems. The goal of this research is to develop methods that will improve the process of testing object-oriented systems. Specifically, emphasis is given to improving the level of testing of methods because the level of method testing is generally considered inadequate. Algorithms are included that identify the set of methods, both interobject and intraobject, that should be tested for a given system. These algorithms are implemented as a part of an automated testing system that derives a framework for the testing of methods. This system includes the automatic generation of test drivers to facilitate the testing. It captures the results of tests for the purposes of reuse for future system maintenance. This framework provides the software engineer who is testing a system a mechanism to determine the level of method coverage that has been achieved in the testing process.

CHAPTER 1. INTRODUCTION

Testing is done to uncover errors in order to instill confidence that a program is operating properly. We view the testing procedure as a part of an integrated process that should: 1) reduce unnecessary testing, 2) help prevent errors from recurring, and 3) uncover errors. Traditionally, testing is conducted after the code has been completed. This practice often leads to a long recode-retest and retest-recode cycle. This cycle can cause unneeded cost and waste of time.

After a program is developed, a programmer often tests what the program does rather than what it should do. Testing the functionality of a program must be planned and designed prior to coding. Programs are tested using various testing techniques to uncover errors in order to produce high quality programs. However, software developed with different software paradigms requires different testing techniques. The features of the paradigm itself determine the choice of and/or the need for different testing techniques.

Successful testing can help to reduce maintenance of software. Since software maintenance is an ongoing process, the more successful the software testing the easier the software maintenance. A large number of errors detected during development indicate that less maintenance is needed. Testing, if done properly, facilitates the timely delivery of systems. In contrast, unorganized and 'last

minute' testing may delay prompt system delivery. Effective testing requires accurate planning. Testing is more costly and time consuming if not carefully planned. In other words, testing is not synonymous with debugging. Testing is performed with the intention of finding errors that lead to faults. Debugging is applied to localize errors in order to rectify them. In [Pressman97] Parnas indicates that testing cannot prove the absence of errors. It can prove that errors are existent in the code.

1.1 Testing Methodologies

There are two general types of testing methodologies: 1) white box testing, 2) black box testing. Testing techniques have been developed to implement these methodologies. Some testing techniques are applicable to both white box testing and black box testing while others are specific to one but not the other. Usually both black and white box methodologies are applied when testing a program.

Howden defines black box testing as testing the program without regard to the internal structure of the code [Howden78]. Programmers or testing teams must select black box testing if the input-output sequence of the program does not adhere to the specification [Howden95]. Advantages of black box testing are: 1) no constraint on input, and 2) rate of error detection is usually high. Disadvantages of

black box testing are: 1) error detection is possible but finite and not as effective as white box testing, and 2) dependence on the completeness and validity of the specifications [DeMillo87]. Black box oriented testing techniques include functional testing and random testing. In functional testing, programs are tested over different input classes. Each function of the system is tested. Advantages include high probability of finding an error, and the possibility of using a valid and invalid input. One drawback of this test is that in some cases it requires extreme case testing.

In [Pressman97], extreme case testing is defined as a set of input data that lies on the boundary of a class of input data or which generates output that lies on the boundary of a class of output data. In functional testing, a test team must generate test output over possible output classes. Also, the team must test the performance of each function to see if it adheres to the expected functionality in the requirement definition document. Howden suggests the use of data abstraction to detect functional errors [Howden95]. Function abstraction is recommended after using data abstraction since it detects design or requirement errors while data abstraction detects few functional errors.

Random testing is a black box strategy in which an input value is selected at random from the possible input domain. For a specific test case, if the number of errors detected increases, then the input test case was a good choice of input test data

domain. However, Myers calls it the poorest strategy [Myers79]. Disadvantages of the random test include the dependence on the completeness and validity of the specifications.

White box testing is a logically driven test of the program where the test is based on prior knowledge of the internal structure of the code. The techniques are easy to implement and not as expensive as the black box testing techniques. Advantages of this method of testing are: 1) error detection is high and occurs frequently if errors exist in the software, 2) simplicity in implementing the approach, 3) prior knowledge of the program helps in comparing actual output with expected output. Disadvantages of this method include: 1) difficult test data generation since this type of testing depends largely on the internal structure of the code, 2) in certain white box testing techniques the test team tests what the program is **doing** rather than **what it should do!** [Pfleeger98], and 3) failure to detect the absence of functions that were specified but not included in the code.

White box oriented testing techniques include path testing, statement coverage, and branch coverage. In path testing, every path is executed at least once by choosing input data that triggers the path execution. Path testing is a must-do test in newly developed software. Path testing is used to detect calculation errors, missing path errors [White87], path selection errors, and dead or unreachable code.

If the module is large, path testing efficiency decreases dramatically. Drawbacks of this technique are: a) inability to uncover missing functions [White87], b) inability to detect specification defects or initialization errors, c) difficulty in detecting, and diagnosing interface errors, d) time consuming, e) complex [Pressman97], f) some existing errors cannot be uncovered, and g) the percentage of error detection is low. With statement coverage, every statement is executed at least once, and with branch coverage, every decision statement is exercised at least once.

The black box and white box techniques have been used in the following testing phases: 1) unit testing where each individual unit of code is tested to verify that its functionality meets the expected specification. Usually after each unit of code is completed, unit testing is performed. 2) integration testing where sets of units of code are tested and their cross-references and interfaces are checked 3) system testing where the overall system is tested with respect to the requirement specification document 4) function testing to test that the performance of functions are as specified in the requirement definition document. In this test, comprehensive testing is conducted to check if the actual software may contain more functions than those specified in the requirements. 5) performance testing to verify the nonfunctional requirements, i.e., speed, response time, accuracy, security, and reliability. 6) acceptance testing to verify that the system features are in accordance

with the defined requirements. The acceptance test includes the alpha test, which is conducted in house; the beta test, which is conducted at the customer site; and the benchmark test in which the customer prepares a set of test cases to be performed.

7) installation testing which involves the user testing the system for compliance with expectations.

Numerous techniques are applied in integration testing. These techniques implement both black box and white box testing methodologies [Pfleeger98]:

1) Top Down: Each main module is tested individually by first applying the unit testing techniques and then the interface testing technique. Advantages of this technique are there is no requirement for a module driver, and there is no need to apply system testing and integration testing. Disadvantages are the need for a stub and the difficulty of developing test data [Pfleeger98].

2) Bottom Up: The leaf modules of the system are individually tested. Advantages of this technique are that choosing test data is not that difficult [Pfleeger98], and no earlier version of the system can be released until the last module is tested [Pfleeger98]. Disadvantages include the requirement of a module driver, the delay of major error detection until the end of test, the difficulty in using this test when concurrence and timing are a major concern [Pressman97], and late detection of errors at the top level.

- 3) **Modified Top Down:** Modules are tested at all levels before an integrated system test. The drawback of using this technique is the need for a module stub and a module driver.
- 4) **Big Bang:** All modules are merged for testing. Drawbacks are difficulty in detecting interface errors, module stubs and module drivers are required, and difficulty in discovering the module causing the error.
- 5) **Sandwich Testing:** Top down and bottom up strategies are incorporated into one testing scheme. The choice of testing techniques depends on the level of interface and functionality of the module. This test possesses advantages of both top down and bottom up techniques. One drawback is the unit test cannot be performed before the integration test.
- 6) **Domain Testing:** Test data for a group of programs is chosen to uncover certain types of errors. The idea is to divide the input into a set of domains called subdomains. Each subdomain has input points that trigger path executions. Domain testing is predicate oriented. Each predicate, i.e., $<$, $=$, $\leq$, $\geq$, $\diamond$, and $>$ has different interpretations. Domain testing is used to prove the correctness of the specification and to interpret predicates based on the input variables. It is also used to reveal path selection errors [Pressman97]. Advantages include detection of faulty logic and detection of math overflow. When domain testing is used on the

specification, i.e. data flow, it is known as black box testing. When it is used on the implementation, i.e. control flow, it is then known as white box testing. Disadvantages include the need for many reliable test points, problems in choosing test data when the program number contains number of input variables, and the need for testing methods since domain testing tests for path selection errors only.

Other testing techniques include mutation testing, equivalence testing, and partition analysis. Mutation testing is a metric to measure the adequacy of the test data. In mutation testing, test data is applied to the original program P and its mutants P_i . Test data is used to reveal if errors are absent from the code in concern. This type of test can be beneficial in detecting dead code, matrix calculation errors [Pressman97], coincidental correctness, and domain errors. The idea behind mutation testing is basically to alter the code in incremental form and then classify faults generated with each altered statement [DeMillo87]. One major disadvantage of the test is that it generates a large set of mutants. [Howden78] has amended this testing technique by introducing weak mutation and strong mutation techniques. Equivalence partitioning divides the input space into equivalence classes, each of which is a case.

Partition analysis applies a combination of structural and functional testing methods to reveal undetected domain and computation errors. This testing technique

is important since one can break the specifications into subspecifications and intersect them with the input domain, thus yielding a test case (partition). Later, evaluation of these partitions can be done to see how close the actual implementation is from the specification. Partition analysis facilitates the input and output compatibility [Pfleeger98].

1.2 Automated Testing Tools

Automated testing tools are a form of data and program analyzers which help programmers to locate errors and fix them. The main reason to use automated testing tools is to reduce cost and time effectively. Their use and application have become a crucial part of today's testing procedures. Automated testing tools are grouped into two major categories, static and dynamic.

Static Analyzers: Static analyzers analyze the code prior to execution. Static analyzers are designed to detect uninitialized variables, dead code, variables used but not defined, variables defined but not used, data flow errors, and to determine if a variable used in a procedure is local or global. Static analyzers include: 1) code analyzers, 2) sequence checkers, 3) data analyzers, and 4) structure analyzers. A special case of a static analysis method is symbolic execution. This is an approach in which input variables are assigned symbols instead of values as input and the output variables are expressed in terms of these symbols [DeMillo87]. Advantages

are: 1) useful for proving program correctness (only reasonable size programs), 2) effective for detecting mathematical, and logical errors, and 3) best results when applied to structured programs. The problems with symbolic execution are: 1) difficulty in creating the assertions, and 2) lengthy and time consuming. Testing with symbols is as good as testing with a large set of cases as in the other testing techniques [Pfleeger98].

Dynamic Analyzers: Dynamic analyzers study the behavior of the program as it is executed. Dynamic analyzers include: 1) program monitors: a program that monitors the state of other programs by displaying its behavior upon either input or output, 2) test probes or counters: statements that are inserted inside a program which report the number of times certain control construct has been executed, and 3) break points: a forced stop points to enable the tester to evaluate the current status of the program behavior, input, and output.

1.3 Testing for Object Oriented Programming

Within the last decade, the use of object-oriented languages (OOL) has been established as a programming method with great possibilities. An object is an entity that is composed of data and procedures. The procedures, referred to as methods, implement the operations on the object data. Each object has a state, an identity, and a behavior. The definition of the type of object in OOL is a description of its

capabilities. A class is a grouping of objects that share in common similar features and actions. An object of a class is an instance. A message is a request for an action and it is composed of argument(s) and a selector. A method is a specific action to reply to a message.

OOL has numerous powerful features. First, inheritance is where a subclass 'legally' uses everything in its superclass. Inheritance is viewed as an addition, i.e., adding a method to an object in a class that differs from the methods in the super class. Inheritance is either one to one (single) or one to many (multiple), and it facilitates reusability if not encourages it [Murphy94]. Inheritance allows an object in the subclass to override the need of multiple instances of a method. Other forms of methods application by a subclass are override or substitution. In the case of multiple inheritance, a class is permitted to have more than one superclass [D'Souza94]. Multiple inheritance increases the possibility of code sharing and also increases the possibility of conflicts.

Second, data abstraction is a concept that allows the encapsulation of the data and the procedures that acts upon the data together in one syntactical unit. Data abstraction improves the usability of data structures, and thus reduces the writing of the same modules in different systems.

Third, information hiding is a powerful feature used on components of the object to reduce unnecessary interfaces of objects and utilizes the concept of data abstraction.

Fourth, dynamic binding is an OOL concept that facilitates the compilation of the software and validates the integrity of the system. By allowing dynamic binding, polymorphism can be implemented. A message is sent to different objects and responded to by different methods. Dynamic binding permits adding new objects of a certain class without having to modify existing code. The major drawbacks of dynamic binding are the cost of runtime and the difficulty involved in implementation [Chidamber91].

We note that a subclass may add a new method, add a class variable, or add an instance variable. A subclass also inherits from the superclass. An object acts as a sender by sending a message with or without argument(s). The selector selects the method by which the message is sent to invoke a response from the other object that is the receiver. A method cannot call a method, although in imperative languages a procedure can call a procedure. Objects only send messages via methods. In OOL, methods cannot be separated from objects. A protocol is a stream of messages to which an object responds.

Public variables provide the interfaces with other objects. Private variables are comparable to data in imperative languages. Likewise, methods in OOL are comparable to procedures in imperative languages. In a given class, there are variables that are shared among the class of objects. Instance variables are a subset of private variables. All instances have the same number of variables but may or may not have same values. However, they use the same methods to respond to a message. Choice of the methods is based on the method selector. The difference in response by two different instances is the difference of their value of instance variables. The object-oriented paradigm is based on program abstraction. A class defines the actions of its instances in response to a message. A class generally links a method with a selector to search for a method to respond to [D'Souza94]. All instances in a class share the same methods. Selection of different methods by two instances is due to differences in value of their instance variables.

Imperative testing techniques have yielded some breakthroughs in object-oriented (OO) testing; however, many researchers and practitioners emphasize the need for a uniform OO testing process [Hoffman89, Collefello90, DeMillo91, Murphy92, Askit92, Rettig91, Jorgensen94, Binder94, McGregor94, Chidamber91, D'Souza94, Holub93]. Also, there have been difficulties in implementing OO testing

theories [Hoffman89, Collefello90, Askit92, Rettig91, Binder94, McGregor94, Murphy94, Chidamber91, D'Souza94, Holub93, Singh93, Parrish93, Xiaoping93].

There is a growing need for a testing assistance for object oriented programs [Hoffman89, Collefello90, DeMillo91, Cheatham90, Gannon81, Jorgensen94, Binder94, Poston94, McGregor94, Arnold94, Harold92, Marick95, Murphy94, D'Souza94, Sametinger90, Hughes96, Devanbu96] Although the research done in this area has been productive, most test tools are difficult to use and hard to implement [Lieberherr93, Fayad93, Jorgensen94, Chaar93, Parrish93, Xiaoping93].

Programs written using imperative languages are tested using imperative testing techniques. Programs written using OO languages are tested by using imperative testing techniques and object oriented testing techniques. These OO testing techniques are typically imperative testing techniques either injected with a few procedures which test certain OO features or they are pure OO testing techniques that are one dimensional, i.e., test one feature or are particular to one system only. Hence, there is a need for a comprehensive OO testing approach for OO programs.

1.4 Problem Definition

There is a need for testing assistant to facilitate more productive OO testing. Limitations to the current work have been cited by numerous practitioners and

researchers [Hoffman89, Collefello90, Askit92, Rettig91, Binder94, McGregor94, Murphy94, Chidamber91, D'Souza94, Holub93, Singh93, Parrish93, Xiaoping93].

One crucial aspect of testing OO code is the testing of all methods in one object and the testing of all methods outside the object that communicate with the object. Testing assistance to ensure that all methods are tested is one issue in testing object oriented programs that has not been addressed sufficiently. Testing all the methods in an object, the interfaces of all methods between objects, and the inheritance between parents and inheriting objects should result in higher error detection at an early stage.

Techniques and frameworks exist to test each object individually but not exhaustively. By exhaustively we mean not only a unit test of the object but also interface test and inheritance test of all forms, single (one to one), multiple (one to many), and multilevel (parent class is a derived class by itself). There is a lack of a framework that ensures that objects are tested exhaustively at the object level by testing the object as a unit, testing the interfaces of methods between the object and other objects, and testing its inheritance. The testing of all methods requires the definition of algorithms to identify tests needed for all methods. Based on the algorithms, a testing assistant is developed to generate test drivers for all methods. This approach provides assistance that helps ensure method coverage is achieved.

Programs developed in a given software paradigm are more efficiently tested by testing techniques that share the same scope of features and language orientation used to develop the program. For example, applying a path testing strategy to a C⁺⁺ program will not work in the same way as when applied to a C program, unless the code is modified. Imperative testing techniques are not adequate to test OO programs. A testing technique is considered adequate if, and only if, once it is exercised with an adequate test data set, the probability of revealing errors is very high [Bergadano96]. A test data set is adequate if it triggers the execution of statements which lead to fault occurrence, thus revealing an error, keeping in mind that an error may cause more than one fault [Bergadano96]. The adequacy of testing is hard to determine unless all errors have been uncovered. However, even though a program does not fail any of the tests and all detected errors are corrected, there is no guarantee of the absence of errors.

Test adequacy is a challenging matter; it is simplified by the existence of methods and theories that help determine the amount and type of testing needed [Howden95, Bergadano96]. A second consideration is the number of testing techniques needed. In testing, increasing the number of tests and testing techniques does not necessarily mean more errors will be revealed. The selection of the proper

testing technique(s) remains the most significant element, and consequently it affects the number of testing techniques selected.

1.5 Overview of research

Review of previous work done in the object-oriented testing indicates that attention has primarily been devoted to the development of either testing techniques, or to testing methodologies [Hoffman89, Collefello90, DeMillo91, Cheatham90, Gannon81, Jorgensen94, Binder94, Poston94, McGregor94, Arnold94, Harold92, Marick95, Murphy94, D'Souza94, Sametinger90, Hughes96, Devanbu96]. The goal of the research is to develop a methodology that improves the object oriented testing process by identifying the suite of tests needed for comprehensive methods coverage. The process includes the following steps: 1) analyzing the code, and 2) devising components of the testing assistant, 3) generating the testing assistant, 4) exercising the testing assistant, and 5) analyzing the results.

Chapter 2 reviews the related work in object oriented testing. Chapter 3 presents the fundamentals for an OO testing framework. Chapter 4 describes the functionality of the testing assistant. Chapter 5 is devoted to the architecture of the testing assistant. Chapter 6 contains case studies to which the testing assistant was applied. Finally, contributions, and conclusions are presented in Chapter 7.

CHAPTER 2. RELATED WORK

Numerous research initiatives have been devoted to testing object-oriented code. Most of the methodologies are focused on either black box or white box testing. Most of the work focuses on testing theories or testing frameworks. Few testing techniques are presented. Testing theories are a presentation of a logical and finite methodology to handle certain type of testing. A testing framework is a test plan that is used as an aid in transforming a testing theory into a testing technique. A testing technique is an implementation of a theory in a technical and procedural form.

We divide the previous research into four groups: 1) a single testing strategy, 2) testing frameworks with theories, 3) implementation of testing strategies, and 4) assessments of testing strategies

2.1. Single testing strategies

Research that focuses primarily on single testing strategies is found in [Hoffman89, Arnold94, Harold94, Wong95, Korel96]. [Hoffman89] discussed unit testing and [Arnold94] presented work on testing object oriented code at the 1) unit level, 2) subsystem level, 3) process level, 4) domain level, and 5) cross-domain level. [Harold94] presented a base class test that involves testing the interaction between classes that have no parents. [Wong95] analyzed and presented the effects

of reducing the size of the test set on fault detection. [Korel96] used fault seeding to generate test data for Pascal programs.

Imperative unit testing methods have been proposed for testing methods for objects in [Hoffman89]. Hoffman's practical approach for module regression testing reduces the cost of test maintenance, development, and execution. The approach included the following principles: 1) develop test code as part of the system, 2) improve cost effectiveness of the test, 3) automate long and time consuming routines, 4) standardize the automization, and 5) test one module at a time.

In [Murphy92] a testing methodology for object oriented systems is presented which is based on issues to be considered at the time of test. These issues are interface, cluster, unit, object, inheritance testing. He provided an initiative for a standard methodology. He summarized pending and difficult issues to be solved without offering a solution.

In [Cheatham90] the areas that must be addressed in OO testing were identified. Difficulties in developing and testing OOP were addressed in [Askit92] with emphasis on design and design evaluation. However, they identified that poor design led to problems and hence requires rigorous testing. They identify problems related to object interactions.

In [Lieberherr93] a test plan design was introduced as in [Rettig91, McGregor94]. The test type was of an incremental type as in [Harold94]. [Lieberherr93] defined and used propagation patterns that are syntactic elements that define a group of C⁺⁺ programs from which members are selected by submitting a class dictionary graph. A class dictionary graph is used to select the C⁺⁺ program since it describes the structure of the object in depth. At each of the software development stages, i.e. design, analysis, and programming, a list of a family of classes and their has-a or is-a definitions are used to describe all objects of the class in detail. No conclusion was made about the difficulty of using the technique with large object oriented programs and the amount of time it requires, since the class dictionary graph and the propagation patterns have to be created for each family of related classes.

The Shlaer-Mellor Object Oriented Analysis Method was implemented in [Fayad93], and the efficiency and cost effectiveness of this method were discussed. It is not a testing methodology but is an analysis method related to OOP and beneficial for OOP evaluation. It provides systematic ways of recognizing objects in a system by analyzing abstract data types. Three formal models are built: information, state, and process. [Fayad93] discussed the limitation of the Shlaer-Mellor method by demonstrating the limitation of methods available to specify

requirements. Shaler-Mellor methods may result in the **wicked problem**. [Fayad93] referenced the Ramamoorthy wicked problem which occurs when the ‘specification is intertwined with the design’, which is a problem resulting from misunderstanding the purpose of each of the following requirements and design stages.

The DAISTS system in [Gannon81] was based on module testing and test cases definition. It is based on the idea that once the system reads in a formal algebraic specification of the module to be tested, it will immediately supply the right test data set for certain test. The system is limited in that, if algebraic specifications are unavailable, then modules testing cannot be done.

In [Rettig91], a method is proposed for helping developers choose the proper testing methods. He suggested the ‘planning ahead’ mentality. He introduced test manager, which is a Smalltalk class that can determine how to respond to a finite set of testing messages. Drawbacks of the systems are 1) writing test procedures is time consuming, and 2) lack on interaction with the user and software ([Rettig91] calls this a technical problem).

Most recently, [Jorgensen94] proposed five levels of object oriented testing. The most important contributions were the Method/Message Path (MM-Path), the Atomic System Function (ASF), and threads. [Jorgensen94] defines the MM-path as a chain of method execution joined by messages. While [Jorgensen94] defines the

ASF as an input event proceeded by a set of MM-paths and completed by an output event. [Jorgensen94] defines threads as a series of method execution connected by messages in the object network. However, threads differ from MM-paths in the sense that the smallest thread has at least one ASF and hence an ASF has at least one MM-path. For a certain class, and by using the above three methods, attributes can be represented in MM-paths, ASFs, and threads. Therefore, error detection is easier once the test is conducted since the five levels of OO testing have the potential to detect errors that can escape unit testing. Experimental results were not given for their work.

In [Binder94] the need for testability for OO features, such as encapsulation, inheritance, and polymorphism, was stressed. As testability factors were defined with the case of imperative language programs, [Binder94] defined the testability factors for OOP. They include: 1) completeness which means that all features of the system are specified in the specification document and there are no "to be determined" parts, 2) traceability which includes the record keeping of which part of the software implements a certain specification and which specification implements a certain requirement, 3) currency involves the fact that the system implemented must be based on the specification 4) separation of concerns which is related to controllability and observability where a software components is

independent if it is controllable, and 5) control strategy refers to a technique that measures the level of control embedded in its interfaces. The control strategy has two types of packaging: collaboration packaging which refers to classes taking part in a responsibility, and architectural packaging which refers to the tasks allocated to classes and how their runtime interfaces are structured. [Binder94] also showed the complexity effect as well as the scope effect for the encapsulation metric, inheritance metric, polymorphism metric, and complexity metric. He stressed interoperability as a key issue in developing a test tool.

In [Poston94] automated testing from object models is discussed. A team applied the Object Modeling Technique (OMT) model in reusing and preparing common objects models to perform automated testing. The OMT model consists of a dynamic model, object model, and functional model. OMT requires the analysis of requirement, design of the system, design of the modules, coding, test cases definition, test case development, running the tests, finally evaluation of software and tests. A Customer Service System (CSS) was developed using the OMT models components 1) object model, 2) dynamic model, and 3) functional model. The modeling tool and the test case generator are powerful components. The system cuts down the cost of test case generation by reusing common components. There is no unique method that assists OO testing with regard to unit testing, integration testing,

or system testing. However, the emphasis was to show that some existing modules or objects could be reused as a part of developing an automated testing tool for object-oriented code.

In [McGregor94], an integrated development / testing process and a criteria for testing models were introduced. The Criteria for Testing Models was defined as Correctness, Completeness, and Consistency. Correctness is defined as the accuracy of analysis entities in semantically defining the reality of those entities. In other words, can these analysis entities be used in the later versions of following models? Completeness means that the model must be understandable to experts as an overall view of certain approved information within its scope. There should be no room for discrepancies in representing relationship of entities in one portion of the model and another portion of the same model. Consistency is based on the relationship among components in the model. A consistent model has representations in one part of the model that are correctly reflected in other portions of the model.

[McGregor94] also followed the steps of [Poston94] in implementing the OMT to devise a technique for an overall system model. A major contribution was the identification of levels of testing. [McGregor94] projected three levels of testing: 1) class testing, 2) cluster testing, and 3) system testing. They define class testing as

similar to unit testing of the imperative approach in concept but not in payoff. The payoff of class testing is higher since classes are more complex than procedures. Also, class testing serves as the first level of integration testing in object-oriented code. They define cluster testing as the test of interaction of the instances of the class in the cluster, which is defined as a grouping of related classes. [McGregor94] defines system testing as functionally testing the system if it meets the requirements originally specified by the user.

[McGregor94] advocate that system testing in object-oriented code is the same as in the imperative approach. The only difference though is the test case selection and implementation. They projected a complete panorama of test case selection and implementation as: 1) functional, 2) structural, and 3) interaction. [McGregor94] introduced the functional test cases as test cases developed by evaluating the class's specification and defined structural test cases as test cases developed by evaluating each separate method's implementation. Interaction test cases were defined as the interface between two cooperating classes. The developer should create the test cases only if it is necessary or as part of the contract. Finally some discussion on test drivers, test execution and validation was provided; however, their work contains no automization of concepts.

[Arnold94] presented results from testing object-oriented code at different levels. The recommended levels of testing are: 1) unit level, 2) subsystem level, 3) process level, 4) domain level, and 5) cross-domain level. [Arnold 94] defined unit level as unit test, subsystem level as superclass test, process level as each UNIX process execution, domain level as client-server type relation test, and cross-domain level as PC to a mainframe. [Arnold94] stressed the problem encountered in C⁺⁺ that makes testing difficult. They projected three aspects of objects that must be tested: 1) behavior, 2) consistency, and 3) processes. [Arnold94] refers to behavior as specification, consistency as standardization of methods implementation in developing an automated testing tool, and processes as the client-server model implemented in the C⁺⁺ format. The tool created a test driver that exercises the input class and its interfaces. The utility carried out a black box technique and left white box testing at the discretion of the developer. They later stressed the need for reuse of working modules in a test tool. The system lacked a static code analyzer that could have supported the test drivers.

An incremental class testing technique, presented in [Harold94], summarized two approaches to class testing. The first approach utilizes the hierarchical structure of classes that is related via inheritance, hence testing only certain methods and not all methods of all subclasses, while the other approach involves complete retesting

of each method in all subclasses. The idea is based on reusing the information for a parent class to test a related cluster and incrementally updating the information to facilitate the subclass testing.

[Harold94] described an OO testing history. Simply, a testing history links each test case with the attributes being tested. A newly created subclass inherits the parent's testing history, and it is incrementally updated to differ from the parents, resulting in a testing history for the subclass. [Harold94] claimed two benefits from this approach. One is the reuse of the parent's testing history while the other is the reduction of overhead cost in retesting of all subclasses. [Harold94] used a traditional imperative technique, i.e., unit testing to conduct the base class testing. However they used an incremental technique in testing subclasses that copies the testing history of the parent's class to the subclass and later updates it if necessary. The system was implemented using two routines 1) automatic determination of how much retesting is needed for a subclass, and 2) aid in the actual subclass testing. Implementation was program-based which is based on data flow testing. Data flow testing involves exercising all procedures by passing the data and monitoring the process. It also involves changing data values via procedures to evaluate the interaction between all procedures that are related. The technique saves time in test

case selection and test implementation. No indication was given as to how well the technique works with larger sets of classes or larger OO systems.

[Wong95] presented the effects of reducing the size of a test set on fault detection. [Wong95] injected faults into a program to develop variant programs from the original one with the intention of cost reduction in the size of the test set. This process was called test set minimization [Wong95]. It was not based on the individual tester time spent on each test case nor on the time needed to traverse each test case. [Wong95] test set minimization was done by exercising a routine which computes the minimal subset.

[Wong95] defined fault detection effectiveness of a test set when applied to a program under test as the ratio of detected faults over the test set of all injected faults. [Wong95] conducted experiments and presented the results as follows :

1) when block coverage is between (50-55)% there is no test set minimization, 2) when the test set is between (80-85)% and (90-95)% block coverage, there is a size reduction, 3) when two sets have the same block coverage for a given program, it is not assured that a reduction of size necessarily reduces effectiveness, and 4) faults detected by many test cases are 'likely' detected after the minimization process. However, faults detected by few test cases are less 'likely' detected after the

minimization process. The research was not done on a large set of programs and no indication was made about the effect of this study on OO programs.

[Korel96] used a different approach of program testing by automization of the testing process with the injection of assertions with the purpose of test cost reduction. The automated test data generated system called TESTGEN supported programs written in subset of Pascal. [Korel96] inserted the assertions as boolean formulas, i.e. logical expressions using and, or, and the not operators, or as executable code to detect errors. Source code is read, and the assertions are generated from the source code or inserted using a chaining approach. [Korel96] used the study to compare their approach with existing methods of test generation. TESTGEN was used to generate the assertions and the test set was used for testing for faults. [Korel96] showed that assertions can be used for automated test generation. The type of fault the test set generation process helps uncover was not studied.

2.2. Testing frameworks with theories

Research that involves testing framework with theories includes [Collofello90, Murphy92, Chetham90, Askit92, Liberherr93, Jorgensen94, Binder94, Poston94, McGregor94, Marick95, Offutt96, Hughs96, Voas95, Khan95, Howden95, and Bergadano96]. [Collofello90] proposed a framework for testing

software practice. [Murphy92] presented a general testing methodology for OO programs. [Chetham90] discussed and identified areas that must be addressed in OO testing. [Askit92] emphasized the importance of design and its effects on testing. [Liberherr93] introduced an incremental test that utilizes propagation patterns and class dictionary graphs. [Jorgensen94] proposed five levels of OO testing. [Binder94] discussed and explained the testability factors in OO programs. [Poston94] used an automated testing model that tests the object and the functions of the object. [McGregor94] introduced a testing process criterion for testing frameworks. [Marick95] discussed the requirement for class requirement catalogs where tests are stored. [Offutt96] introduced a semantic model of program faults. [Hughs96] presented a semantic algebraic testing system for OO called Daitish. [Voas95] emphasized the concept of software testability. [Howden95] stressed the detectability of faults in programs and how it affects software trustability. [Bergadano96] proved that testing can be improved by means of inductive program learning.

In [Collofello90], a testing methodology was proposed as a general framework. The framework was not implemented as a testing technique. The framework covered many issues such as: 1) testing levels definitions as well as the difference of entry and exit criteria, 2) mapping of different testing techniques to

different testing levels with the recommendation of testing techniques that are applicable for each level, and 3) metrics identification.

[Marick95] stressed the need to implement plans in order to conduct testing on objects and classes. He preferred tests to be rerun between classes and derived classes since the 'smallest reasonable subsystem is the class.'. [Marick95] presented the class requirement catalog that places the requirements to be tested in a catalog file. The class requirement catalog contains 1) object use requirement, 2) member function requirement, and 3) state machine requirement. This method requires less work since the class requirement can be copied from the class catalog to the routine test requirement checklist. The test requirement checklist is a list of steps to be considered when testing code or specification. The test requirements are sorted by the availability of code or specification or both. Just as in [Harold94], [Marick95] indicates how to build the derived class requirement catalog from the base class requirement catalog. For each inherited member function, the integration requirements are copied, leaving only the domain shift requirements. Integration requirements involve the manipulating or the changing of a data member. The domain shift requirements are mainly the preconditions and post conditions in an operation.

Overriding and new (pure) member functions get no requirements. Each changed member function must be rewritten later. New data members are handled as any variable. For each new member function (pure), the integration requirements is built in the same format for the original member functions. For the overriding member functions, the integration requirements is built in the same format for the original member functions but, if they differ in functionality than the base class, they are marked as NEW. [Marick95] prefers retest of methods if they are overridden, test of methods that are pure, and no test if they are inherited provided that they were earlier tested in the parent's class. No implementation of the OO testing framework was described.

[Offutt96] introduced a semantic model of program faults and removed any ambiguity revolving around the definition of an error and fault. [Offutt96] highlighted the fact that fault injection into a program inflicts changes to the program and is depended on the syntactic nature of the fault. On the other hand, if the fault is “naturally” within the program, then the number of corrections needed to fix the fault best describe the syntactic nature of the fault. Fault seeding is used on methods to categorize possible errors that were not detected from applying the new testing techniques on the methods. Fault seeding is the injection of errors in a program and/or mutating a program with the intention of generating predetermined

faults [Wong95, Offutt96]. [Offutt96] also introduced two other terms: syntactic size, and semantic size of a fault. They are respectively the count of lines of code or statements that need to be altered to produce a correct program and the size of the subdomain where the output mapping is incorrect. [Offutt96] stressed fault seeding as injecting the faults into programs to test the efficiency of the test strategy or to compare the strategy to other strategies and mutants as set of programs that are close to the original program. [Offutt96] claims that mutants that are small syntactically but large semantically only generate noise. [Offutt96] also discussed the coupling effect which means that complex faults are coupled to simple faults such that test data that uncovers all simple faults in a certain program will detect most complex faults.

[Hughes96] presented a semantic algebraic testing system for OO called Daistish. Daistish is a tool that creates test drivers for programs written in C⁺⁺ to duplicate objects for comparison purposes. Daistish uses ADT structures, utilizes Guttag-style algebraic specifications to scan all input files, and creates parse trees for axioms and test vectors. Daistish is composed of aliases, signature, axioms, and test vectors. Aliases are alternate names to types, signatures files describe functions used in axioms and test vectors, axioms are named expressions that will be true if the axiom is matched by the parameters, and test vectors are sample instantiation of

the types used by the axioms. It also allows compositional creation of test data. The evaluation of Daistish was from “anecdotal evidence” from students. [Hughes96] designed a C⁺⁺ version of Daistish to test programs; however, it does not support objects being passed by value or by reference.

[Voas95] emphasized the concept of software testability. They introduced verification as defined in the IEEE standards glossary of software engineering terminology which is “process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase”, and introduced testability as defined in the IEEE standards glossary of software engineering terminology which is “the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met”. The claim is that proper testability aids in testing rather than testing “blind” and that it increases “confidence of correctness”. They used testability to evaluate the quality of programs. They suggest two different ways to reduce the test set as in [Wong95]: first, by choosing a test set that has a high possibility of revealing faults or second, by designing programs that have high possibility of failure. However, [Voas95] prefers the second way whereby programs should infect itself with damages. The program must contain constructs that are error causing. They asserted that designing programs for

testability eliminates the “too little, too late problem”, i.e., faults uncovered at the verification stage are costly and too late to be changes. They used domain-to-range ratio model to specify software testability on every function in a program and also encouraged the use of sensitivity analysis, execution analysis, infection analysis, and propagation analysis to improve designing software to increase its testability. The study was a conceptual study but was not specific to OO testing.

[Khan95] presented a framework for developing OO programs and discussed the overall benefits of such a paradigm and framework. The structured programming paradigm and the object oriented paradigm were compared. The framework involved steps for developing and maintaining object oriented programs. The framework assists a programmer to shift code development techniques from the structured programming to an object oriented programming. They did not present any OO testing recommendation.

[Howden95] discussed software trustability analysis. Dependability of testing methods and their reliability were also analyzed. [Howden95] emphasized the detectability of faults of program and how it affects software trustability. He defined detectability, D , of a certain method as the probability a fault will be detected in a certain program by that specific method. [Howden95] defined trustability as $T = 1 - (1 - D)$ where $1 - D$ is a risk factor of the program containing

faults. More analysis to detectability and fault density was conducted via means of comparison between partition testing and random testing. Generation of test cases for random testing and functional testing was also compared with the cost benefit factor leaning to the favor of random testing. [Howden95] contained theoretical concepts transformed into an algorithmic solution. Finally, [Howden95] showed the relationship between testability and detectability, and trustability and reliability.

[Bergadano96] proved that testing can be improved by means of an inductive program learning routine. He redefined the reliability of programs as in [Howden95] with discussion of the concept of test case adequacy. The relationship of test case adequacy and the set of program mutants was also cited. [Bergadano96] defines Induction as the “inference from examples to programs; test case generation is from programs to input values”. So a series of inductive conclusions of programs from examples can generate test cases. An algorithmic routine was presented as an inductive procedure of any certain program. [Bergadano96] showed examples of complex cases but with no references to the suitability of this study in the OO paradigm. No supportive data was presented of how useful this inductive procedure can be to practitioners.

2.3. Implementations of testing strategy

Implementations of testing strategies are featured in [DeMillo91, Devanbu96, Kung95a, Kung95b]. [DeMillo91] developed a software testing / debugging tool called Mothra to perform system verification. [Devanbu96] developed a system called Aria that generates testing and analysis tools. [Kung95a] introduced a test strategy for OO programs via object relation diagram. [Kung95b] introduced another model that is an extension of other existing state models that allow the object state testing for OO programs.

In [DeMillo91], an automated software testing / debugging tool was developed using Mothra, an oracle toolset developed and used for automated system verification. Mothra implements the technique of program mutation to carry out the actual tests. Although it is written in standard C and designed for imperative languages, the idea of designing a toolset, which uses 'mutant' programs to discover errors, is applicable for testing OOP programs.

[Devanbu96] developed a system called Aria. Aria was generated by an interactive system called Geno that generates testing and analysis tools based on a tool called Reprise. It accepts abstract semantics graph representation for C and C⁺⁺. The graph representation is specified for Reprise and is migrated into the Geno system. The processor, called Genii, translates them into graph dictionary and feeds

them to Aria which translates them to code via traversals, imperatives, and expressions.

[Devanbu96] used Aria to specify a cyclomatic complexity estimator tool, called Cyco, to compute the McCabe Metric for Complexity. Cyco computes approximate path conditions called Synpatco. It uses Aria to derive dependence graphs. The study lacked a 'rich collection of data structures' and it did not modify any of the Abstract Semantic Graphs (ASGs), a graph representation of a program, during the analysis stage. ASGs are an abstract syntax tree (AST) with embedded semantic information.

[Kung95a] introduced a test strategy for OO programs via a Object Relation Diagram (ORD) derived from C^{++} source code. Their test strategy calls for finding the least amount of cost and effort needed to create test stubs to test an OO program. The strategy is to break the member functions and methods into modules connected via edges that represent relationship. A routine determines the Object Relation Diagram (ORD), a reverse engineering tool that obtains relationships between classes from C^{++} source code for either an acyclic digraph or a cyclic digraph. Entities in an acyclic digraph are tested by the topological sorting of set of classes based on the interdependency of relationship of classes while entities in a cyclic digraph cannot be tested in the same manner. However, they provided an algorithm

that solves the second case by transforming via an iterative procedure the cyclic digraph into an acyclic digraph and applying the topological sorting to do the test. The transformation is equipped via breaking a cycle routing which removes some of the directed associated edges temporarily to form an acyclic digraph. The strategy lacked the detail and availability of how structure, function, object state, and data flow testing must be done. Also no supplementary data was used as a proof of how successful the strategy is on large programs.

[Kung95b] introduced another model that is an extension on other existing state models that allows the object state testing for OO programs. It concentrated on the behavior of class objects in OO programs which highlighted the following steps for object state testing : 1) developing a testing model, 2) validate the model, 3) select the test strategies, 4) choose test criteria, and 5) generate test cases. [Kung95b] developed a model, Object State Diagram (OSD), which supports inheritance and OO function oriented object states. The idea was to categorize the OSD into either Atomic Object State Diagram (AOSD) and Composite Object State Diagram (COSD). The former represents the states of a data in a class and their transitions while the latter represent the validation of object state and the transition of the dynamic behavior for the class. An AOSD is a Finite State Machine (FSM) diagram, i.e., states connected by transitions and have a start and a goal state. A

COSD has 1) defined parts, which represent states and transition which is an AOSD, 2) aggregated parts, which represent the state behavior of an object, and 3) inherited parts, which represent the dynamic behavior of inherited structures including their states and transitions. After the OSD is created, the OSD testing procedure composed by [Kung95a] is applied. The testing procedure searches for input to trigger test cases for object state, object transition, condition transition, and interacting transition. The object state testing is done in a bottom up fashion, the object and condition transition testing are done using test drivers creation and test case generation, while integration transition testing is done via identifying common transitions and operations between AOSDs/COSDs and generating test cases for them.

2.4. Assessments of testing strategies

Assessments are found in [DeMillo95, Berard96, Maring96, Doan95, and Offutt98]. [DeMillo95] discussed the interpretation of the Frankl-Weyuker model for test criteria selection. [Berard96] emphasized the importance of testing with additional planning and user interaction. [Maring96] stressed the need for additional testing methodologies and strategies for OO programs. [Doan95] showed the effect of user involvement in testing. Finally, [Offutt98] identified problems in academic research and education which affect testing.

[DeMillo95] discussed the interpretation of the Frankl-Weyuker Model for test criteria selection. [DeMillo95] applied mutation analysis on C and stressed the need to use decision coverage. [Berard96] emphasized the problem that testing is done with ‘too little planning by people with too little training’ and encouraged management involvement. He provided a framework for testers and managers. In [Berard96] the most notable contribution to testing is that “a tester’s job is to uncover problems, not prove that a product works”. [Maring96] explained that testing has not gotten enough researcher attention and added that more testing methodologies and strategies are needed especially for reusable OO components.

[Doan95] showed results of how important it is to involve users in the testing of OO programs. [Offutt98] emphasized the difference between testing theories and testing techniques that are practiced in industry and taught in academia.

Limitations of the related work include:

- 1) few presentations of experimentation.
- 2) limited coverage of inheritance
- 3) no inheritance complexity
- 4) lack of frameworks to generate testing assistants

Table 2.1 illustrates the comparison between existing automated object oriented methodologies or strategies. The main goal of this research is to develop

techniques that ensure that all methods (inter and intra) are tested within an object. The goal is to help ensure the testing of all methods individually in each object, the interfaces of all methods between objects, and the inheritance between parents and inheriting objects.

Table 2.1 Efficiency Comparison of Models

Feature	Hoffman	Collofello	DeMillo	Murphy	Askit, Bergmans	Lieberherr Xiao	Fayad,Hawn Roberts,Klatt	McMullin Hamlet	Alkadi
1	*		*				*	*	*
2			*			*	*		*
3			*						*
4									*
5									*
6	*		*		*	*	*	*	*
7	*		*		*	*	*	*	*
Feature	Rettig	Jorgensen Erickson	Binder	Poston	McGregor Korson	Arnold Fuson	Harold, Fitzpatrick	Marick	Alkadi
1	*	*		*		*			*
2	*						*	*	*
3					*	*	*		*
4			*		*		*	*	*
5			*			*	*		*
6	*	*		*	*	*	*		*
7	*	*		*		*	*		*
Feature	Wong, Horgan	Korel, Al-Yami	Offutt, Hayes	Hughes, Stotts	Devanbu, Rosenblum	DeMillo, Mathur	Voas, Miller	Berard	Alkadi
1		*	*	*	*				*
2		*	*	*					*
3		*	*	*	*		*		*
4									*
5									*
6	*	*	*	*	*				*
7		*	*	*	*				

(Table cont'd)

Feature	Neilsen	Maring	Khan, Al-Aali	Kung, Gao	Hsia, Chen	Bergadano Gunetti	Howden Huang	Doan Paton	Alkadi
1									*
2									*
3				*	*	*	*		*
4									*
5				*	*				*
6				*	*	*	*		*
7				*					*

Features - A star designates inclusion while N/A indicates not available

- 1) improves the potential of thoroughly method testing.
- 2) maintains a history file that can be revisited at any time as a reference for future occurrence of errors.
- 3) standardization in which the six classes in the data structure have the same method naming convention and the same style of code.
- 4) inheritance test (TIC) is not only done for the single/multiple inheritance but also for the multi level inheritance.
- 5) implementation of a test plan based on a customized testing process.
- 6) generates test drivers that can be reused.
- 7) generates test drivers selectively.

CHAPTER 3. FUNDAMENTAL COMPONENTS OF AN OBJECT-ORIENTED TESTING ASSISTANT

In this research, we identify testing techniques which should be performed when testing an object oriented system. Section 1 is devoted to the presentation and discussion of object level test while section 2 is devoted to the presentation of inheritance test within classes.

Testing an OO program is a challenge. The difficulty enhanced by the powerful features of the OO paradigm. The difficulty in testing is increased with the use of polymorphism, data abstraction, inheritance, and dynamic binding. Inheritance, for example, exists in different forms: 1) single where a subclass inherits from only one parent, 2) multiple where a subclass can inherit from any of its parents, and 3) multilevel where a subclass can inherit from any of its parent which is a subclass of another parent class. There is a need for a testing techniques that cover these features. Currently, tests are frequently conducted in separate procedures, i.e., unit testing, integration testing, class level testing, and system testing.

The approach defined in this research incorporates unit testing, interface testing, inheritance testing, and method coverage. The tests work at the system level if they are broken down to different classes, i.e., functionality is broken down independently by class. The framework will not support cluster or system

(integration) testing as they are available in other testing techniques [Jorgensen94, Harold94]. Methods should be tested separately before object testing. The goal is to test each method in an object. For every method m in object O , there must be a message that triggers the execution of this method. **Method Coverage** in the object-oriented paradigm is what statement coverage is in the imperative model. It is the execution of methods until all methods have been exhaustively covered.

The fundamental components of an object oriented testing assistant are:

1. Object Level Test (OLT)

- A. For unit testing, all methods are tested in one object. (OT)

- B. For interface testing, all communication calls between objects via methods are covered (IT).

2. Inheritance Test (TIC)

- A. all pure methods are tested

- B. all overridden methods are retested

- C. all inherited methods are uncovered.

3.1 Object Level Test (OLT)

A tester can implement an already existing technique, such as path testing, statement testing, or branch testing, to conduct unit testing. The unit testing technique plus the different testing strategies (path, control, branch, and statement

testing) are applicable in object testing in OO as well as other imperative testing techniques.

The Object Level Test (OLT) tests that every instance of an object in a class communicates properly with objects in the same class or in the class scope of communication. OLT is related to unit testing; however, the type and range of errors that can be discovered from OLT are more diverse than the errors discovered from the imperative unit testing. OLT includes two kinds of tests: 1) object testing (OT), and 2) interface testing (IT). The interfaces between objects and methods of two different objects are tested in Interface Test (IT). The object containing the method and the object calling the method are used to test the method that links these two object execution sequences.

3.1.1 Object Test

The object test (OT) tests every method in one object of a class. Interface testing then follows, in which each of the object interfaces is tested. Interface testing is necessary because errors that occur at this level of code infrastructure cause other errors to trigger that can result in other faults at the system level. The tester may have difficulty tracing the original cause of errors if they are not detected at the interface testing stage. Hence, error localization becomes costly in both time and effort.

Figure 3.1 illustrates object testing. We declare a GeometricObject class and the class Two_D_Object which inherits from GeometricObject. The GeometricObject is a class that allows the user to draw, manipulate, and erase a geometric object based on values entered by the user. The method Two_D_Object::Enter_Input() requires testing in order to achieve OT.

```

class GeometricObject
{
    public:
        void Draw();
        void Erase();
        void Move(int h, int v);
        GeometricObject();
        GeometricObject(int t, int l, int b, int r)
            {top = t; left = l; bottom = b; right = r;};
    protected:
        int left, top, bottom, right;
};

:
:

#include <iostream.h>
class Two_D_Object : public GeometricObject
{
    public:
        int fillPattern;
        void findCenter(int&, int&);
        Two_D_Object();
        Two_D_Object(int t, int l, int b, int r, int fill):GeometricObject(t,l,b,r)
            {fillPattern = fill;};
        void Enter_Input();
        void Draw();
};

GeometricObject square(top,left,bottom,right);
square.Draw();

:
:

```

Figure 3.1 Unit Testing Example

3.1.2 Interface Test

After the object has been tested at the object level, it should be tested at the class 'integration' level. A class is integrated by calls from its objects to methods in other objects in the same class. Object interface testing is similar to integration testing in the imperative model. Imperative models are broken down into modules per each level as in (fan-in) and (fan-out) designs [Pfleeger87], while OO systems are isolated as objects. We test the interfaces between objects and methods of two different objects in this test. The object containing the method and the object calling the method are used to test the method that links these two object execution sequences.

Figure 3.2 illustrates interface testing. We declare `dogs`, `main_class`, `spotted_dogs`, and `unspotted_dog` classes where the last two classes inherit from the parent class `dogs`. The `dogs` class allows the creation of a dog of a certain breed, height, weight, and color. It also allows the breed to be displayed. The classes `spotted_dogs` and `unspotted_dogs` allow the creation of the same type of dogs but with color spots. The `main_class::main_prog()` method requires testing to test for interface errors between `main_class` and `dogs`. One object of one class is calling a method in another object of a different class. `Main_class` is calling

snoopy.show_breed(), and snoopy is of type spotted_dog hence it calls upon the parent class method dogs::show breed().

```
class dogs
{
    public:
        dogs(char *breed, int height, int weight, char *color);
        void show_breed(void);
        dogs();
    private:
        char breed[64]; int height; int weight; char color[64];
};

// declaration file for spotted and unspotted dogs class
#include <iostream.h>
#include <string.h>
class main_class
{
    public: void main_prog(void);};

class spotted_dogs : public dogs
{
    public: spotted_dogs(char *breed, int height, int weight, char *color, char
*spot_color);
        void show_breed(void); void spot_info(void);
    private: char spot_color[64];
};

class unspotted_dogs : public dogs
{
    public: unspotted_dogs(char *breed, int height, int weight, char *color);
};

    spotted_dogs snoopy(breed,height,weight,color,spot);
    unspotted_dogs bloopy(breed,height,weight,color);
    snoopy.show_breed(); snoopy.spot_info(); getch(); break;
    bloopy.show_breed(); getch(); break;
    :
    :
```

Figure 3.2 Interface Testing Example

3.2 Inheritance Test

3.2.1 Inheritance Test Within Classes (TIC)

The second test is Testing Inheritance within Classes (TIC). This test is based on white box testing. When an object inherits from a parent class, it inherits all methods and data structures, which optimizes code reuse and minimizes code

redundancy. Within OO, the possibilities of 1) inheriting the code and reusing it, 2) inheriting the code and altering it (overriding) or 3) adding separate code (pure) force different testing approaches.

Figure 3.3 illustrates single and multilevel inheritance testing. We declare the OrgEntity class, classes main_class, Company, Division, and Department classes, where the last three inherit from the class OrgEntity. The method main_prog::start_up() requires testing to uncover errors from the single inheritance in Company and parent class OrgEntity, and the multilevel inheritance in department class which inherits from Division and Division inherits from company.

Figure 3.4 illustrates multiple inheritance testing. We declare book, disk, main_class, and bundle classes in which the latter inherits from both classes book and disk. The method main_class::main_prog() requires testing to uncover any errors from the multiple inheritance in bundle and parent classes disk and book.

3.2.2 Inheritance Test Complexity Estimation

In addition to testing the object at the unit level and the interface/integration level, testing the object at the global level, i.e., the range of all possible parents from which the object can inherit methods, is mandatory. We derive an estimator for the number of test drivers necessary, based on the number of inherited methods.

The estimator assumes that inherited methods are not retested and that pure and overridden methods should be tested. It is based on the sum of methods which require testing or retesting. The number of inherited methods in one object is added to the complexity estimation if they were not tested at the parent level. The estimator C_c is defined by:

$$[1] C_c = P + O + I$$

Where

P   # of pure methods

O   # of inherited overridden methods

I   # inherited methods not tested at the parent level

C   Complexity of inheritance testing

C_c is the inheritance test complexity estimation for class c . As an example of estimation of the number of test drivers required, assume there exists a class **C** with two methods and three member objects: O_1 , O_2 , and O_3 . Let O_1 have three pure methods and one overridden method, let O_2 have four pure methods and two overridden methods, and let O_3 have two pure methods and no overridden methods. The following formula determines the estimation:


```

class OrgEntity
{char name[25];
int employee_count;
public: OrgEntity(char *s, int ec);
OrgEntity();
int number_employees() { return employee_count; }
char *org_name() { return name; } virtual int office_party() = 0;
void budget(OrgEntity& oe);
};

// The declaration file for Multiple Derived Classes
#include <iostream.h>
class main_prog
{public: void start_up(void);};
class Company : public OrgEntity
{
public: Company(char *s, int ec); virtual int office_party();
};
class Division : public Company
{
    public: Division(char *s, int ec); virtual int office_party();
};

class Department : public Division
{
public: Department(char *s, int ec); int office_party();
};
Company business(company, code);
business.budget(business);
getch(); clrscr(); break;
Division section(div, code);
section.budget(section); getch(); clrscr(); break;
Department branch(dept, code);
branch.budget(branch); getch(); clrscr(); break;
.
.

```

Figure 3.3 Single & Multi-level Inheritance Example

```

class book
{
    public:
        book(char *title, char *author, int pages);
        book();
        void show_book(void);
    private:
        char title[64], author[64];  int pages;
};

class disk
{
    public:
        disk(char *name, int sides, int tracks, int sectors_per_track, int
bytes_per_sector);
        void show_disk(void);
    private:
        char name[64];
        int sides;
        int tracks;
        int sectors_per_track;
        int bytes_per_sector;
        long capacity;
};

#include <iostream.h>
#include <string.h>

class main_class {
    public:
        void main_prog();
};

class bundle : public book, public disk
{
    public: bundle(char *title, char *author , int pages, float capacity, float
price);
        bundle();    void show_bundle(void);
    private: float price;
};

    bundle jurassic_park(title,author,pages,capacity,price);
    jurassic_park.show_bundle();
    getch();
    clrscr(); break;
    .
    .
    .

```

Figure 3.4 Multiple Inheritance Example

$$[1] \quad C(O_1) = 3 + 1 + n \text{ (not tested in parent)} = 4 + n$$

$$[2] \quad C(O_2) = 4 + 2 + n \text{ (not tested in parent)} = 6 + n$$

$$[3] \quad C(O_3) = 2 + 0 + n \text{ (not tested in parent)} = 2 + n$$

Then for the class C , $C_C = \Sigma C_i = C(O_1) + C(O_2) + C(O_3) = 4 + 6 + 2 + 3(n)$, where n is the number of methods not tested in the parent. This value helps to plan the testing procedure after the design and coding are completed.

In summary, the OLT and TIC testing techniques are the fundamental building blocks that are used to define the testing framework. Successful and effective testing require planning. To plan a test, a testing framework, which covers object testing, object integration testing, and inheritance is needed. Methods in an object have to be individually tested to determine whether they comply with the functionality specified in the software requirement specification and design document. The interface between objects covers the integration at the system level, and hence must be tested.

The testing framework helps to ensure that method coverage is done. Object tests, interface tests, and inheritance tests are required. In Chapter 4 we describe the functionality of the testing assistant which incorporates these fundamental building blocks.

CHAPTER 4. FUNCTIONALITY OF THE ASSISTANT

4.1 Overview

The automated testing assistant requires: 1) defining the components of the testing assistant, 2) implementing the components of the testing assistant, 3) exercising the testing assistant, and 4) analyzing results. Figure 4.1 gives an overview of the assistant.

Figure 4.1 Test Plan Definition

Sections 4.2, 4.3, and 4.4 describe the design of the automated testing assistant.

4.2 Definition of the Testing Assistant

The testing assistant is designed when the objects, classes, and methods, are designed. After the source code analyses, both static and dynamic, a specific testing

Figure 4.2 Framework for the assistant

assistant is defined. The testing techniques, described in Chapter 3, are the nuclei of the testing assistant. These tests are implemented in a testing assistant which creates the test drivers. The user does not view the internal creation of the test driver. The system compiles, links, and runs the test driver.

4.3 Components of the Testing Assistant

We define an algorithm to compute the number of test drivers needed for the OLT and TIC, and an algorithm to generate the number of test drivers needed for the OLT and TIC. The testing assistant will automatically create the test drivers, compile them and run them dynamically if the user desires. [Hoffman89, DeMillo91,

Rettig91, Jorgensen94, Binder94, Poston94, McGregor94, Harold92, Arnold94, Marick95, Murphy94, Hughes92, Devanbu96, Kung95a, Kung95b] have called for the development of such a testing assistant in OO. However, we were unable to find a toolkit that incorporates all these tests.

4.4 Production of the Testing Assistant

Figure 4.2 illustrates how the computation of the test drivers is done. For each object, the test drivers needed for OLT and TIC are added to the total number of test drivers needed. Estimation of the number of test drivers needed to conduct a system test helps the tester in planning the testing process. Also, computing the number of test drivers needed helps in comparing this number against the number of test drivers that are actually generated from the automated testing assistant. The number may vary if selective generation is done. However, if automatic generation is done, there should be a match in the number of test drivers computed from the OLT and TIC test drivers computation algorithm and the number of test drivers generated from the OLT and TIC test drivers generation algorithm.

4.4.1 OLT & TIC Test Driver Computation Algorithm

For each system S , store the number of objects in S in N . Then for every object O_i store the number of methods in M . Add the number of methods to be unit tested to the array U , add the number of methods in different objects to test their

interface to the array I, and add the number of inherited methods to be test/retested into the array H based on : 1) test the method if it is pure, 2) retest the method if it is overridden, and 3) skip if the method is inherited unaltered.

```

Let   S      :      Object Oriented System
      O      :      Finite Set of Objects where  $O = \{O_1, O_2, \dots, O_N\}$ 
      N      :      Number of objects
      M      :      Number of methods in one object
      TO     :      Array [1..p] of Finite set of tests for an object
       $\alpha_1$  :      Finite set of tests for S based on components one and two
      U      :      array [1..p] of integer;[stores # of unit tests per object]
      I      :      array [1..p] of integer;[stores # of interface test per object]
      H      :      array [1..p] of integer;[stores # of inheritance test per object]

Repeat for every system S
Begin
    1) acquire # of objects into N
    2) for k := 1 to N do          { For every  $O_k$ }
        begin
            3) acquire # of methods in  $O_k$  into M
            4)  $U[k] := U[k] + M$ ; {all methods must be unit tested in object}
            5) For each interface between  $O_i \gamma O_j$ ,  $0 < i, j \in [N, i \gamma j]$ 
            6)  $I[k] := I[k] + 1$ ;
            7) For each inheritance between  $O_i \gamma O_j$ ,  $0 < i, j \in [N, i \gamma j]$ 
                begin
                    8) If inheritance type is = pure then  $H[k] := H[k] + 1$  {pure methods}
                    9) else If inheritance is = overridden then  $H[k] := H[k] + 1$ ; {overridden}
                       Else skip; {since inherited methods are not to be tested again}
                end;
            10)  $TO_i := U[k] + I[k] + H[k]$ ; {All unit + interface+ inheritance test for an object}
            11)  $\alpha_1 := \alpha_1 + TO_i$ ;
                end;
        end;
    End. {repeat}

```

Proof

$\forall$ object O_i in a system S , the set of methods M , such that $M = \{m_i, m_{i+1}, \dots, m_n\}$ has to be unit tested as well as each interface between one object and the other object(s) via methods. Obviously a method that is inherited should not be retested but the methods that are inherited but overridden must be retested. Furthermore the pure methods have to be tested. We then add the number of tests to compute the test drivers required for each object and we call it α_1 .

The algorithm computes the number of test drivers needed. Once we implement the algorithm then we can use the testing assistant to generate the test drivers and hence, we can run all the tests to complete all the OLT and TIC.

4.4.2 System Computation Algorithm

The following algorithm computes the number of test drivers needed for the testing of the entire system. If other requirements besides the object specific requirements are tested, then this algorithm counts the number of test drivers that will be needed to complete the test. The data to compute the number of test drivers for the additional requirements must be supplied by a system engineer. Each test driver added is not generated by the automated testing assistant, but adding them to the total number of test drivers needed help the tester in estimating how much testing to be done. A separate testing assistant or individual test driver may be developed

to test each additional system requirement such as time, concurrency, parallelism and networking.

For each system S, store the number of additional requirements in tests.f. Then set the value of α_2 to the value of α_1 . For every additional requirement and every object, add the predefined specific function based methods into Tests.A. Add the number of methods to be tested to the TO_i . From the OLT and TIC algorithm we add the value of α_1 to α_2 and we get the α_3 ,i.e., the total number of test drivers required for the entire system.

Let   S   :   Object Oriented System
       O   :   Finite Set of Objects where $O = \{O_1, O_2, ..., O_N\}$
       N   :   Number of objects
       M   :   Number of methods in one object
       TO   :   Array [1..p] of Finite set of tests for an object
        α_2 :   Finite set of tests for S based on specific functions
       Tests   :   record
                  F : integer; [number of features]
                  X : array[1..F] of strings; [features and functions to be tested]
                  A :array[1..F] of integer; [additional # of tests needed per x]
                  end;

Repeat for every system S

Begin

- 1) acquire # of features into Tests.F
- 2) set α_2 to the value from previous algorithm
- 3) for j := 1 to F do { For every function such as concurrence, parallelism, etc..}

Begin

  - 4) Input name of feature into Tests.X[j]
  - 5) for k := 1 to N do   { For every O_k }

begin

    - 6) set TO_k to the value from previous algorithm
    - 7) acquire # of methods in O_k that are involved in Tests.X[j]

```

      8) Tests.A[k] := Tests.A[k]+1;{methods tested for a specific feature}
      9) TOi := TOi + A[k];
      10) α2 := α2 + TOi;
      end;
    End;
  End. {repeat}

```

Proof

∀ object O_i in a system S, the set of methods M, ∈ M = {m_i, m_{i+1}, ..., m_n}, must be tested according to a predefined specific function or features. We then add the number of tests to compute the test drivers required for each object, called α₂. The algorithm has to compute the number of test drivers needed for the specific additional requirements for the system under test to complete the system's overall test drivers requirements.

The count computed from both algorithms enables the tester to prepare and plan the proper testing needed. After applying the first algorithm for computing the test drivers, the counts computed from the first algorithm should match the number of test drivers generated by the OLT and TIC test drivers generation algorithm given in section 4.4.3.

4.4.3 OLT & TIC Test Drivers Generation Algorithm

Following is an algorithm that generates test drivers either selectively for one specific method or automatically for all methods. For every object in the system, the user is prompted to either selectively generate a test driver for one method specified

by the user or to let the automated testing assistant automatically generate all test drivers for all methods. This algorithm generates test drivers within the object to implement the Object Test (OT).

4.4.3.1 (Object Test) test drivers generation algorithm

Let   S   :   Object Oriented System
      O   :   Finite Set of Objects where $O = \{O_1, O_2, \dots, O_N\}$
      M   :   Number of methods in one object
      N   :   Number of objects
      Choice: integer;

Repeat for every system S

Begin

1) acquire # of objects into N

2) for k := 1 to N do   { For every O_k }
   begin

3) prompt the user to select between:

   I) selective generation of one test driver

   II) automatic generation of all test drivers

4) if choice = I then
   begin

     prompt for class name

     prompt for method name

     prompt for any parameters

     prompt for file name where class declaration resides

     write test driver using the following algorithm:

     begin

        a) initialize a dynamic array of code to null characters

        b) move the #include library statements necessary for
            the code to run and link to external libraries to the
            upper portion of dynamic array of code. The
            following is a list of needed libraries in C⁺⁺ so that
            any C⁺⁺ program can run.

            <iostream.h>

```

        <fstream.h>
        <iomanip.h>
        <stdlib.h>
        <stdio.h>
c) append the #include statement where class
   declaration resides to the dynamic array of code.
d) create an instantiation of an object from the user
   entered class and append it to the dynamic array of
   code
e) append to the dynamic array of code the statement
   which invokes the method selected for test
f) append the dynamic array of code with "(" at the
   end of the previous line of code
g) for (number of parameters entered) do
    begin
        l) append the previous line in the
           dynamic array of code with the
           parameter name
    end
h) append the dynamic array of code with ")" at the
   end of the previous line
i) for every line in the dynamic array of code
    begin
        l) move the contents of the dynamic
           array of code to the test driver
    end
j) write the test driver to a file called testprog.cpp
end {algorithm}
end {choice}
else
begin
    prompt for file name where class declaration resides
    for each method  $M_i$  in  $O_k$  write test driver using the following
    algorithm:
    begin
        a) initialize a dynamic array of code to null characters
        g) move the #include library statements necessary for
           the code to run and link to external libraries to the
           upper portion of dynamic array of code. The

```

following is a list of needed libraries in C++ so that any C++ program can run.

<iostream.h>
<fstream.h>
<iomanip.h>
<stdlib.h>
<stdio.h>

- b) append the #include statement where class declaration resides to the dynamic array of code.
  - c) create an instantiation of an object from the user entered class and append it to the dynamic array of code
  - d) for every method in the object do
    - begin
      - I) append to the dynamic array of code the statement which invokes the method selected for test
      - II) append the dynamic array of code with "(" at the end of the previous line of code
      - III) for (number of parameters entered) do
        - begin
        - a) append the previous line in the dynamic array of code with the parameter name
        - end
      - IV) append the dynamic array of code with ")" at the end of the previous line
    - end {method}
  - f) for every line in the dynamic array of code
    - begin
      - I) move the contents of the dynamic array of code to the test driver
    - end
  - e) write the test driver to a file called testprog.cpp
- end {algorithm}

```

        end {choice}
5)    compile test driver
6)    link test driver
7)    run test driver for that method  $M_i$  in  $O_k$ 
End. {repeat}

```

4.4.3.2 (Interface Test) test drivers generation algorithm

The following is an algorithm that generates test drivers either selectively for one specific method or automatically for all methods. For every object in the system, the user is prompted to either selectively generate a test driver for one method specified by the user or to let the automated testing assistant automatically generate all test drivers for all methods. This algorithm generates test drivers to handle the interface between objects via methods implement the Interface Test (IT).

```

Let   S      :    Object Oriented System
      O      :    Finite Set of Objects where  $O = \{O_1, O_2, \dots, O_N\}$ 
      M      :    Number of methods in one object
      N      :    Number of objects
      Choice:    integer;

```

Repeat for every system S

Begin

```

1) acquire # of objects into N
2) for k := 1 to N do      { For every  $O_k$  }
   begin
       3) prompt the user to select between:
          I) selective generation of one test driver
          II) automatic generation of all test drivers

       4) if choice = I then
          begin
              prompt for class name

```

prompt for method name

prompt for any parameters

prompt for file name where class declaration resides

write test driver using the following algorithm:

begin

- a) initialize a dynamic array of code to null characters
- h) move the #include library statements necessary for the code to run and link to external libraries to the upper portion of dynamic array of code. The following is a list of needed libraries in C++ so that any C++ program can run.

<iostream.h>

<fstream.h>

<iomanip.h>

<stdlib.h>

<stdio.h>

- b) append the #include statement where class declaration resides to the dynamic array of code.
- c) create an instantiation of an object from the user entered class and append it to the dynamic array of code
- d) append to the dynamic array of code the statement which invokes the method selected for test
- e) append the dynamic array of code with "(" at the end of the previous line of code
- g) for (number of parameters entered) do
begin

- I) append the previous line in the dynamic array of code with the parameter name

end

- f) append the dynamic array of code with ")" at the end of the previous line

- g) for every line in the dynamic array of code
begin

- I) move the contents of the dynamic array of code to the test driver

end

- j) write the test driver to a file called testprog.cpp

```

        end {algorithm}
    end {choice}
else
begin
    prompt for file name where class declaration resides
    for each interface between  $O_i \gamma O_j$  ,  $0 < i, j \leq N$ ,  $i \gamma j$  do the
    following algorithm:
    begin
        a) initialize a dynamic array of code to null characters
        i) move the #include library statements necessary for
            the code to run and link to external libraries to the
            upper portion of dynamic array of code. The
            following is a list of needed libraries in C++ so that
            any C++ program can run.
                <iostream.h>
                <fstream.h>
                <iomanip.h>
                <stdlib.h>
                <stdio.h>
        b) append the #include statement where class
            declaration resides to the dynamic array of code.
        c) create an instantiation of an object from the user
            entered class and append it to the dynamic array of
            code
        d) for every method in the object do
            begin
                I)    append to the dynamic array of
                    code the statement which invokes
                    the method selected for test
                II)   append the dynamic array of code
                    with "(" at the end of the previous
                    line of code
                III)  for (number of parameters
                    entered) do
                    begin
                        1) append the previous line in the
                           dynamic array of code with
                           the parameter name
                    end
            end
        end
    end
end

```



```

IV)    append the dynamic array of code
        with ")" at the end of the
        previous line
    end {method}
f) for every line in the dynamic array of code
    begin
        I)    move the contents of the dynamic
                array of code to the test driver
    end
    g) write the test driver to a file called testprog.cpp
end {algorithm}
end {choice}
5)    compile test driver
6)    link test driver
7)    run test driver for that method  $M_i$  in  $O_k$ 
End. {repeat}

```

4.4.3.3 (Inheritance Test) test drivers generation algorithm

The following is an algorithm that generates test drivers either selectively for one specific method or automatically for all methods. For every object in the system, the user is prompted to either selectively generate a test driver for one method specified by the user or to let the automated testing assistant automatically generate all test drivers for all methods. This algorithm generates test drivers within the object to implement the Inheritance Test within Classes (TIC).

```

Let   S      :    Object Oriented System
      O      :    Finite Set of Objects where  $O = \{O_1, O_2, \dots, O_N\}$ 
      M      :    Number of methods in one object
      N      :    Number of objects
      Choice:    integer;

```

```

Repeat for every system S
Begin

```

- 1) acquire # of objects into N
- 2) for k := 1 to N do   { For every O_k }
     begin
  - 3) prompt the user to select between:
    - I) selective generation of one test driver
    - II) automatic generation of all test drivers
  - 4) if choice = I then
     begin
    - prompt for class name
    - prompt for method name
    - prompt for any parameters
    - prompt for file name where class declaration resides
    - write test driver using the following algorithm:
    - begin
      - a) initialize a dynamic array of code to null characters
      - j) move the #include library statements necessary for the code to run and link to external libraries to the upper portion of dynamic array of code. The following is a list of needed libraries in C⁺⁺ so that any C⁺⁺ program can run.
        - <iostream.h>
        - <fstream.h>
        - <iomanip.h>
        - <stdlib.h>
        - <stdio.h>
      - b) append the #include statement where class declaration resides to the dynamic array of code.
      - c) create an instantiation of an object from the user entered class and append it to the dynamic array of code
      - d) append to the dynamic array of code the statement which invokes the method selected for test
      - f) append the dynamic array of code with "(" at the end of the previous line of code
      - g) for (number of parameters entered) do
     begin

```

I)    append the previous line in
      the dynamic array of code
      with the parameter name

      end
h) append the dynamic array of code with “)” at the end
  of the previous line
i) for every line in the dynamic array of code
  begin
      I)    move the contents of the dynamic
            array of code to the test driver

      end
h) write the test driver to a file called testprog.cpp
end {algorithm}
end {choice}
else
begin
  prompt for file name where class declaration resides
  for each inheritance between  $O_i \gamma O_j$  ,  $0 < i, j \leq N$ ,  $i \gamma j$  do the
  following algorithm:
  begin
    a) initialize a dynamic array of code to null characters
    k) move the #include library statements necessary for
        the code to run and link to external libraries to the
        upper portion of dynamic array of code. The
        following is a list of needed libraries in C++ so that
        any C++ program can run.
        <iostream.h>
        <fstream.h>
        <iomanip.h>
        <stdlib.h>
        <stdio.h>
    b) append the #include statement where class
        declaration resides to the dynamic array of code.
    c) create an instantiation of an object from the user
        entered class and append it to the dynamic array of
        code
    d) for every method in the object do
        begin
          I)    append to the dynamic array of

```

```

                                code the statement which invokes
                                the method selected for test
II)    append the dynamic array of code
        with “(“ at the end of the previous
        line of code
III)   for (number of parameters
        entered) do
        append the previous line in the
dynamic array of code with the parameter name
        end
IV)    append the dynamic array of code
        with “)” at the end of the
        previous line
        end {method}
f) for every line in the dynamic array of code
    begin
        I)    move the contents of the dynamic
              array of code to the test driver
        end
    g) write the test driver to a file called testprog.cpp
    end {algorithm}
    end {choice}
5)    compile test driver
6)    link test driver
7)    run test driver for that method  $M_i$  in  $O_k$ 
End. {repeat}

```

The automated testing assistant facilitates rigorous testing by making the test drivers available to the user. The testing assistant involves two tests. The first test which is the object level test OLT includes OT and IT. The second is the inheritance within classes, called TIC. The testing assistant allows the user to generate a test driver for one specific method or it generates test drivers for all methods. The testing assistant provides all test drivers necessary to ensure that all methods are

tested. The components of the testing assistant are the algorithms. These algorithms are used to implement the testing assistant. The testing techniques presented in chapter 3 are implemented in the testing assistant. The testing assistant is implemented and tested. The architecture of the testing assistant is given in chapter 5.

CHAPTER 5. ARCHITECTURE OF THE TESTING ASSISTANT

5.1 Overview

Based on the specification of the framework for the testing assistant defined in section 4.2, we have designed and implemented a testing system that incorporates object testing, interface testing, and inheritance testing. The goal is to derive the automated tool to create a test driver per method for each kind of test. We have developed a toolset OOTA, which is the implementation of the testing framework that incorporates the proposed tests. OOTA offers selective generation of a test driver for a specific method or automatic generation of test drivers for all methods.

5.2 Overview of the System

OOTA was developed with the purpose of 1) transforming the steps of the methodology into procedures, 2) incorporating or reusing the tool in testing of other systems in the future, and 3) analyzing errors for future reference.

5.3 OOTA Implementation

OOTA runs under Borland C⁺⁺ running under Windows 95/ Windows 98 operating system. OOTA is a menu driven interactive system. In each test type, the user is prompted to supply information as required by the OOTA. OOTA is composed of four classes: 1) object, 2) interface, 3) inheritance, and 4) results. Input varies from one program choice to another. However, the output is standardized. The output is a test driver for 1) OLT (OT and IT), and 2) TIC (single, multiple, and

multiLevel) inheritance. An array is used for storing the code needed to create the test driver. In selective generation of a test driver of a particular method, the user must enter the class name, the method name, any parameters, and the file name where the superclass is defined. For automatic generation of all test drivers for all methods, the user enters only the file name where the class is defined, and the system will automatically generate all test drivers for all methods. The input requirements differ as the menu choices differ. The system also maintains history files where errors are stored cumulatively for classification reasons. A user can print the history file at any time using the print program declared in the file results.h. The data file is stored in a file called results.dat. See Figure 5.1 for the components layout of the system and Figure 5.2 for the main menu screen.

5.4 OOTA test data sets

The OOTA test data set was composed of eighty-eight different runs ranging from textbook code [Stevens93, Jamsa95, Schildt94, and Oualline95], student's C⁺⁺ code, and production software. Each of the test data sets was run and actual output was compared to the expected output. However, tests were ordered, i.e., the object test was administered first, followed by the interface test, and then the inheritance test data set.

Figure 5.1 OOTA Structural Layout

Figure 5.2 Main Menu Screen

5.5 OOTA Internal Structure

The main program provides the user with five options: 1) object test, 2) interface test, 3) inheritance test, 4) print history test result, and 5) exit, shown in Figure 5.2. Table 5.1 shows classes that were used in the main program but defined in separate files.

The file `commtest.h` contains the declaration of the class `tester`, which is used to perform the user data entry for the object test (OT) using the screen shown in Figure 5.3. In the case of selective generation of one test driver for a specific method, the user is prompted to enter the class name, the method name, any parameters for the method under test, and the file name where the class declaration resides.

<u>Object Test Main Menu</u>	
1.	Selective generation of a test driver
2.	Automatic generation of all test drivers
Enter Choice: 2	
Enter file name for the class	: <i>person.h</i>

Figure 5.3 Object Test Main Menu Screen

In the case of automatic generation of all test drivers for all methods in an object, the user is prompted to enter the file name where the class declaration

resides. This declaration file `commtest.h` contains one method called `insert_testdriver` and has a public defined variables of type array called `line_code[80]` where the test driver will be temporarily stored before it is actually written to a file.

The file also contains an array of parameters called `param_name[15]` in case the method has parameters that are acted upon. We set the limit to only five parameters but this limit can be changed as needed. Other private variables in that class include `size`, `class_name`, `method_name`, and `name_file`.

Table 5.1 OOTA Internal Structure

The table content is completely redacted with a solid black box. Only the header information is visible.

The file `inttest.h` contains the declaration of the class `xrefstest`, which is used to perform the user data entry for the interface test (IT). The user is prompted to enter the first class name, the second class name, the called method name, any

parameters for the method under test, the caller object name, the called object name, and the file names where both of the classes declaration reside. This declaration file contains one method also called `insert_testdriver` and has a public defined variables of type array called `line_stmt[80]` where the test driver will be temporarily stored in it before it is actually written to a file. The file also contains an array of parameters called `param_names[15]` in case the method has parameters that are acted upon. We set the limit to only five parameters the limit can be changed as needed. Other private variables in the class include `size`, `class_name1`, `class_name2`, `caller_object`, `callee_object`, `method_name`, and `name_file1`, `name_file2`.

The file `inherit.h` contains the declaration of the class `inherit` which is used to perform the user data entry for the inheritance test (TIC). The user is prompted to enter his/her choice from a menu, shown in Figure 5.4, that offers: 1) estimate inheritance test complexity, 2) single inheritance test, 3) multiple/multilevel inheritance test, and 4) exit. The resulting screens are illustrated in figures 5.4, 5.5, and 5.6 respectively.

OOTA Inheritance Test Main Menu	
1.	Inheritance Complexity Estimator
2.	Single Inheritance Test Menu
3.	Multiple/ <i>Multilevel</i> Test Menu
Enter Choice:	

Figure 5.4 Inheritance Test Main Menu Screen

If the user chooses the first choice, the system prompts him/her to enter the number of pure methods, the number of overridden methods, and the number of inherited methods. Complexity is then computed and then displayed to the screen.

<u>Single Inheritance Test Menu</u>	
Enter file name where Superclass is declared :	<i>faculty.h</i>
Enter class name :	<i>main_class</i>
Enter Method Name:	<i>main_prog</i>
Enter Number of parameters (0 if none) :	<i>0</i>
Enter file name that has the derived class :	<i>Faculty4.h</i>

Figure 5.5 Single Inheritance Test Menu Input Screen

It is based on the formula discussed in section 3.3. If the user chooses the second option from the menu, then the system prompts the user for the class name, the method name, any parameters for the method under test, and the file name where the class declaration resides.

<u>Multiple/Multilevel Inheritance Test Menu</u>	
Enter the 1st file :	<i>person.h</i>
Enter the 2nd file :	<i>working1.h</i>
Enter Class Name:	<i>main_class</i>
Enter Method name :	<i>main_prog</i>
Enter Number of parameters (o if none) :	<i>0</i>
Enter file name that has the derived class :	<i>manager.h</i>

Figure 5.6 Multiple/MultiLevel Inheritance Menu Input Screen

If the user chooses the third option from the menu, the system prompts the user for the number of parent classes, the file names of parent classes if the number of parent classes is not zero, the derived class name, the method name, any parameters for the method under test, and the file name where the derived class declaration resides. This declaration file contains one method called `insert_testdriver` and has a public defined variables of type array called `program_code[80]` where the test driver is temporarily stored in it before it is actually written to a file. The file also contains an array of parameters called `user_param[15]` in case the method has parameters that are acted upon. We set the limit to only five parameters, and the limit can be changed as needed. Other private variables in that class include `size`, `class_name`, `method_name`, `parent_names[4]`, `parent_file`, `parent` and `file`.

The file `results.h` contains the declaration of the class `report`, which is used to perform the user data entry to store the results from all tests. This declaration file contains one method called `assign_values` and has public defined variables of type structure called `report_rec` including the variables `testnum`, `error_type[80]`, `class_name`, `method_name`, and `name_file`. The method will prompt the user to enter the test number, the class name, the method name, file name, and description of error. All values are written to the `report_rec` array before they are actually written to the file `results.dat` on disk. The main program contains all the necessary include files and the declaration files as well.

Table 5.2 illustrates the objects that were instantiated with each of the classes shown earlier in table 5.1. Also the following array variables were declared: code from the array variable test_code from the class tester, stmt from the array variable stmt_code from the class xreftest, and driver from the array variable program_line from the class inherit. A test driver is then created and stored in the appropriate array. The array is then assigned to a file named testprog.cpp. The history results file is also open as input/output and append mode so it may contain cumulative data and is assigned to a file named results.dat. An initialization procedure follows the setting of all arrays, i.e. Object, inheritance, and interface, to null values. The next step includes the assignment of the necessary include statements to the array test_code[I], line_code[I], and driver[I].

Table 5.2 OOTA Classes, Objects, and Purposes

The program displays the main menu as in Figure 5.2. Following the user's choice, it invokes the appropriate method. If the user choice is one, shown in Figure

5.2, then the method `commtest.insert_testdriver();` is invoked. As explained earlier, this method will prompt the user for certain values to be used and stored as lines of codes in the array `test_code.line_code[I]`. In the case of selective generation, and from the class name entered by the user, an object is created and instantiated in the test driver program. It is called `INSTANCE1`, see Figure 5.7. Then the method name is entered with its parameters, if any exist, followed by a null character as a terminator for the test driver code. In the case of automatic generation of all test drivers for all methods, the user enters only the file where the class is defined and an object is created and instantiated in the test driver program. It is called `INSTANCE1`, see Figures 5.3, 5.8, and 5.9 respectively. Then all the methods names are included in the test driver with their parameters entered by the user, if any exist, followed by a null character as a terminator for the test driver code.

```
#include <fstream.h>
#include <stdio.h>
#include <conio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <elevator.h>
main()
{
    Elevator INSTANCE1;
    INSTANCE1.position_floor();
}
```

Figure 5.7 Testprog.cpp test driver – Selective Object Test

Finally, the test driver testprog.cpp is written from the array test_code.line_code[I] and the batch file OOT.BAT is invoked. This batch file compiles and links the test driver. After successful compilation and linking, the user is prompted whether or not they desire to run the test driver. If so, then the batch file COMM.BAT is invoked and consequently the program testprog.exe is executed on the fly. The user then can enter the test number, the error found, the class name, the methods name, the declaration file, and the description of the error. The history file is written to the file results.dat.

```
class person
{
    public:person(char *name, int age);
           show_person();
           showage();
           showname();
           display_age(int age);
};
```

Figure 5.8 person.h Declaration file for class person

```
#include <fstream.h>
#include <stdio.h>
#include <conio.h>
#include <stddef.h>
#include <stdlib.h>
#include <iostream.h>
#include <person.h>
main()
{person INSTANCE1;
INSTANCE1.person(john,44);
INSTANCE1.show_person();
INSTANCE1.showage();
INSTANCE1.showname();
INSTANCE1.display_age(44);
}
```

Figure 5.9 Testprog.cpp test driver – Automatic Object Test

If the user chooses option two, shown in Figure 5.2, the method `crostest.insert_testdriver()`; is invoked, prompting the user for certain values to be used and stored as lines of codes in the array `stmt_code.line_stmt[I]`. From the class name entered by the user, an object is created and instantiated in the test driver program. It is called `INSTANCE1`, see Figure 5.6. Then the method name is entered with its parameters, if any exist, followed by a null character as a terminator for the test driver code. Finally, the test driver `testprog.cpp` get written from the array `stmt_code.line_stmt[I]` and the batch file `OOT.BAT` is invoked. This batch file, compiles, and links our created test driver and this whole process is invisible to the user.

After successful compilation and linking, the user is prompted whether or not he/she desires to run the test driver. If so, then the batch file `COMM.BAT` is invoked and consequently the program `testprog.exe` is executed on the fly. The user then can enter the test number, error type, class name, methods name, declaration file, and the description of the error. The history file is written to the file `results.dat`.

If the user chooses option three, shown in Figure 5.2, the method `inhritst.insert_testdriver()`; is invoked., prompting the user for certain values to be used and stored as lines of codes in the array `programt_line.program_code[I]`.

For the second option, the program follows the same procedure as for Object testing. For the last option, the number of parent classes, is passed from the

inheritst.h to the main program. Based on its value, the test driver is created and lines of code are re-indexed. From the class name entered by the user, an object is created and instantiated in the test driver program. It is called INSTANCE1, see Figure 5.6. Then the method name is entered with its parameters, if any exist, followed by a null character as a terminator for the test driver code.

Finally, the test driver testprog.cpp is written from the array program_line.program_code[I] and the batch file OOT.BAT is invoked. This batch file compiles and links the created test driver. After successful compilation and linking, the user is prompted whether or not they desire to run the test driver. If so, then the batch file COMM.BAT is invoked and consequently the program testprog.exe is executed on the fly. The user then can enter the test number, error type, class name, methods name, declaration file, and the description of the error. The history file is written to the file results.dat.

If the user chooses option four, shown in Figure 5.2, a system call is made to the object file report.exe, which prints the history test data results file to the printer.

5.6 Features of OOTA

OOTA has several features:

- 1) Simplicity since the program is user friendly and menu driven.
- 2) Standardization in which the six classes in the OOTA data

structures section have the same method naming convention and the same style of code as far as user interface is concerned.

- 3) Storage of all errors from all the dynamic runs.
- 4) Estimation of the amount of test drivers needed for inherited methods.
- 5) OOTA offers selective and/or automatic generation of test drivers.

OOTA provides a framework that helps to ensure that appropriate components and interactions are tested by generating code segments that drive the testing process. OOTA facilitates the testing of object-oriented code by incorporating procedures to support object level testing and inheritance testing.

OOTA derives test drivers for OLT and TIC. The tool is productive to use since the OLT and TIC tests are crucial tests. It covers all forms of inheritance, i.e., single multiple, and multiLevel. OOTA identified the possible components. It has the option of either selective generation of a test driver for one specific method or automatic generation for all methods in one object of a class. It contains options that are menu driven and easy to use. OOTA inheritance complexity estimator and the results file are additional features that assist the tester.

CHAPTER 6. TESTING ASSISTANT CASE STUDIES

6.1 OLT (Object Level Testing)

In the following section we demonstrate the OLT test using OOTA. The OLT involves unit testing of all methods in one object, and interface testing which covers all communication calls between objects via methods.

6.1.1 Object Testing

Figure 6.1 represents the test data input elevator.h declaration file. The input data set is a program that services requests of users to go to a certain floor in a building. The number of floors in the building is a constant set to ten. The elevator can move up or down based on the user choice. The object elevator is instantiated and the current position of the elevator is set to floor one. Based on the user request(s), the elevator stores all request(s) as the elevator is moving up or down in the field going. The elevator positions itself at a certain floor if the desired floor is in the same direction as it is headed, even though some further requests exist in the opposite direction. The method MoveToNextFloor will relocate the elevator at the next floor requested and display a message indicating so. The method ChooseMove is the decision maker of whether requests may be served immediately or at a later time. We chose option one from the OOTA menu to conduct an Object test on Elevator.h. The user entered the name of the class, i.e. elevator, the name of the

method, i.e. Position_Floor, any parameters, and the file name where the class elevator is declared, i.e. Elevator.h.

The test driver created by OOTA is shown in Figure 5.7. The test driver is automatically created, compiled, and linked. The test driver contains all the necessary #include statements of libraries of functions required for any C++ program to run. The test driver may be edited and changed if more code or libraries are needed for the compilation or test. Then it may be recompiled accordingly. The test driver also includes the name of the header file where the class definition resides. In Figure 5.7, the header file is <elevator.h>. An instantiation is created from the class by the statement <classname> INSTANCE1, in which INSTANCE1 is an object of the class <classname>. The <classname> is Elevator and INSTANCE1 is the object of Elevator. The <classname> may vary automatically (OOTA generates test drivers for all methods) or selectively (OOTA generates a test driver for one specific method) depending on the user choice. The instantiation is followed by a call INSTANCE1.<methodname> in which the <methodname> is the name of method to be tested. Again, the name of the method depends upon whether the user decides to run the automatic or the selective test driver generation. When the driver is run, the user is given the choice to append to the history file and/or log the error discovered in the history file. By having the test driver generated automatically or selectively, OOTA paves the way for better and more rigorous testing of object

oriented code. The test driver will not locate the errors but can help reveal them if they exist. As shown in Figure 6.1, the error revealed was the **return -1** statement. This error causes the elevator not to service the next request (floor) in the queue. The statement should have been **return 1** and not **return -1**. OOTA by itself did not discover the error, however, the automated generation of test drivers and their subsequent dynamic runs aided in error discovery and increased the potential of better object oriented code testing.

```
// This is the declaration file for class Elevator
#include <iostream.h>
#ifndef ELEVATE1_H_
#define ELEVATE1_H_
const int FLOORSINBLDG = 10;
enum Direction {DOWN,UP};
enum Button {OFF,ON};
class Elevator
{
public:Elevator();
    void MoveToNextFloor();
    Button floorButtons[FLOORSINBLDG+1];
    int currentFloor;
    Direction going;
    void DisplayStatus();
    void WaitForUser(void);
    void Position_Floor();
private:
    int ChooseMove();
};
#endif
Elevator::Elevator()
    currentFloor=1;going=UP;for (int I=0; I < FLOORSINBLDG+1; I++)
    floorButtons[i]=OFF; }
```

Figure 6.1 elevator.h Declaration file for class elevator (figure Cont'd)

```

        int ButtonsPushed();
    {
    {
        for (int t=1; t <= 2; t++) {if (going==UP)
            {for (int I = currentFloor; I < FLOORSINBLDG+1; I++)
                if (floorButtons[i]==ON) return i;going = DOWN;continue;}
            else
                {for (int I = currentFloor; I > 0; I--)
                    if (floorButtons[i]==ON) return i;going = UP;continue;}}

        return -1;}
    void Elevator::MoveToNextFloor()
    {int newFloor=ChooseMove();
    if (newFloor==-1) cout << "Elevator Stopped at Floor " << currentFloor;
        else
            {
                currentFloor=newFloor;floorButtons[currentFloor]=OFF;DisplayStatus();} }
    void Elevator::DisplayStatus()
    {
        cout << "\nElevator is currently stopped at floor " << currentFloor << " heading ";
        if (going==UP) cout << "UP\n";
        else
            cout << "DOWN\n";WaitForUser();}
    void Elevator::WaitForUser(void)
    {getch();} // Freeze the screen until the user types a character.
    void Elevator::Position_Floor()
    {int floor;Elevator e;clrscr();cout << "Please enter a floor Number: ";cin >> floor;
    e.DisplayStatus();e.floorButtons[floor] = ON;e.floorButtons[floor+4]=ON;
    e.MoveToNextFloor();cout << "Please enter a floor Number: ";cin >> floor;
    e.floorButtons[floor] = ON;e.MoveToNextFloor();e.MoveToNextFloor();
    cout << "\nThere you go Sir/Madam, we made it!!!\n";getch();}

```

6.1.2 Interface Test (IT)

Figure 6.2 and Figure 6.3 represent two data input programs billing.h and billwork.h. The billingItem is a class that contains cost and name of item to be sold. The class product inherits from BillingItem and has the field qty_sold which stores the number of items sold. The method product assigns the value of formal parameters to local parameters. The method display shows the value of the field qty_sold. The class service inherits from billing item and has the field manhours as

its own variable. The method service assigns the local parameters values from the formal parameters.

The install class inherits from product and service classes. The method display shows the name, labor, service type, man hours, and cost. The method enter_input clears the screen, prompts the user to enter service type, quantity, hours, price, and instantiate the object inst as an Install type. We chose option two from the OOTA menu to conduct an interface test on billing.h and billwork.h. We entered the name of both classes, i.e. Install and BillingItem, the name of the method, i.e. Enter_Input, any parameters, and file name where the class Install and BillingItem is declared, i.e. billing.h and billwork.h.

The class install has an object declared as inst inside the method install::Enter_input(). The object inst inherits from product and service classes.

The test driver created by OOTA is shown in Figure 6.4. The test driver is automatically created, compiled, and linked. The test driver contains all the necessary #include statements of libraries of functions required for any C++ program to run. The test driver may be edited and changed if more code or libraries are needed for the compilation or test. Then it may be recompiled accordingly. The test driver also includes the name of the header file where the class definition resides.

In Figure 6.4, the header files are <billing.h> and <billwork.h>. An instantiation is created from the class by the statement <classname> video, in which video is an

object of the class `<classname>`. The `<classname>` is Product and video is the object of Product. The other `<classname>` is Install and vcrs is the object of Install. The `<classname>` may vary automatically (OOTA generates test drivers for all methods) or selectively (OOTA generates a test driver for one specific method) depending on the user choice. The instantiation is followed by a call `INSTANCE1.<methodname>` in which the `<methodname>` is the name of method to be tested. Again, the name of the method depends upon whether the user decides to run the automatic or the selective test driver generation. When the driver is run, the user is given the choice to append to the history file and/or log the error discovered in the history file. By having the test driver generated automatically or selectively, OOTA paves the way for better and more rigorous testing of object oriented code. The test driver will not locate the errors but can help reveal them if they exist. As shown in Figure 6.3, the error revealed was the `cin.getline(name,80)` statement. This error causes the salary to be assigned to both local variables the salary and bonus. The statement should have been `cin.getline(service_type,2)` not `cin.getline(name,80)`. OOTA by itself did not discover the error, however, the automated generation of test drivers and their subsequent dynamic runs aided in error discovery and increased the potential of better object oriented code testing.

```

class BillingItem {
    protected:
        char name[25];
        int cost;
    public:
        virtual void display() = 0;
};
class Product : public virtual BillingItem
{
    int qty_sold;
public:
    Product(char *nm, int qty, int cst);
    Product();
    void display();
};

Product::Product(char *nm, int qty, int cst)
{
    qty_sold = qty;
    strcpy(name, nm);
    cost = cst;
}
void Product::display()
{
    cout << qty_sold;
}
Product::Product()
{
}
class Service : public virtual BillingItem
{
    int manhours;
public:
    Service(char *nm, int mh, int cst);
    Service();
    void display();
};
Service::Service(char *nm, int mh, int cst)
{
    manhours = mh; strcpy(nm, name); cost = cst;
}
Service::Service()
{
}
void Service::display()
{
    cout << manhours;
}

```

Figure 6.2 Declaration of billingitem, service, and product

```

// This is the declarations file for the Install class
#include <iostream.h>
#include <graphics.h>
#include <string.h>
class Install : public Product, public Service
{
public:
    Install(char *nm, int qty, int hrs, int cst) : Product(nm, qty, cst), Service(nm, hrs, cst) { }
    Install();
    void Enter_Input(void);
    void display();
};
Install::Install()
{
}
void Install::display()
{cout << "nInstalled ";Product::display();cout << ' ' << name;cout << "nLabour:
";Service::display();cout << " Hours";
cout << "nCost: $" << cost;}
void Install::Enter_Input()
{int okay, user_choice, quant, hour, price; char service_type[64], tmp [2];okay = 0;
while (okay == 0) {window(10,10,40,4);
clrscr();textcolor(YELLOW);textbackground(GREEN);gotoxy(30,4);
cprintf("The Study of virtual base classes");gotoxy(30,5);
cprintf("=====n\n"); gotoxy(24,8);
cputs("[1]. Write a Bill for a Work Order\n");gotoxy(24,9);cputs("[2].
Exit\n");gotoxy(24,19);textcolor(BLINK);
textbackground(YELLOW);delay(200);cputs("Please Enter your choice : ");cin >> user_choice;
switch (user_choice)
{
case 1 : clrscr();
cputs("nPlease Enter the Service type : ");
cin.getline(tmp,2,'n');
cin.getline(name,80); // should have been service_type, hence service_type was never properly
initialized
cputs("nPlease Enter The Quantity : ");
scanf("%d",&quant);
cputs("nPlease Enter the time spent (in hours) : ");
scanf("%d",&hour);
cputs("nPlease Enter the cost : ");
scanf("%d",&price);
Install inst(service_type, quant, hour, price);
inst.display();
getch(); clrscr(); break;
case 2 : okay = 1; break;
default : gotoxy(24,20); sound(848); delay(200); nosound(); cputs("You entered an invalid
input, Please try again..");
getch(); }
} //while}

```

Figure 6.3 install.h Declaration file for class install

```

#include <fstream.h>
#include <stdio.h>
#include <conio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <billing.h>
#include <billwork.h>
main()
{
    Product video;
    Install vcrs;
    vcrs.Enter_Input();
}

```

Figure 6.4 Testprog.cpp test driver – interface test

6.2 Inheritance Test (TIC)

We should test all pure methods, retest all overridden methods, but not test inherited methods. The interpretation of an object inheritance complexity is as follows: for all methods in object I , the inheritance testing complexity is $C_I = 4p + 4o + 2i$ which indicates that a total of ten test drivers are needed. In reality, there are only eight test drivers needed since inherited methods are assumed to be tested earlier in the object test of the parent. Therefore, only pure and overridden methods have to be tested. However, if the parent methods are not tested yet, then this formula will give the user an estimation of how many test drivers are needed, including the parents method.

Figures 5.5, 6.5, 6.6, and 6.7 represent the user input screen and the two test data input program faculty.h, and faculty4.h. Faculty.h is the superclass while faculty4.h is the derived class. The file faculty.h contains the declaration of the class employee where by it contains the field name, home phone, office phone, and manager name. The method employee assigns values of formal parameters to local parameters. The method show_employee displays information about that specific employee. The file faculty4.h contains the declaration class salaried. It inherits from employee, i.e. a case of single inheritance. The method salaried assigns values of formal to local parameters.

```
class employee
{public:employee(char *name, char *home_phone, char *office_phone, char *reports_to);
    employee(); void show_employee(void);
    private: char name[64];char home_phone[64];
            char office_phone[64];char reports_to[64];
};
employee::employee()
{}
employee::employee(char *name, char *home_phone, char *office_phone, char *reports_to)
{
    strcpy(employee::name, name);
    strcpy(employee::home_phone, home_phone);
    strcpy(employee::office_phone, office_phone);
    strcpy(employee::reports_to, reports_to);
}
void employee::show_employee(void)
{
    cout << "Name      : " << name << endl;cout << "Home Phone  : " << home_phone << endl;
    cout << " Office Phone : " << office_phone << endl; cout << "Reports to   : " << reports_to <<
    endl;
}
```

Figure 6.5 Declaration file for employee class

The method show_salaried displays the values for salary, bonus level, and the assistant name. The classes hourly and temporary inherit from employee, and their declaration of data structure is in the same file. The method show_hourly displays

the value for the wage. The method `hourly` assigns the value of the formal parameter of wage to the local parameter wage in the method `hourly`. The method `temporary` also assigns the value of the formal parameter of wage to the local parameter wage in the temporary methods. The method `show_temporay` displays the value for the wage. The class `main_class` contains the method `main_prog` which prompts the user to choose processing a paycheck for either a salaried, temporary, an hourly employee; or displaying information about an employee of any type (hourly, salaried, or temporary). Depending on user choice, the objects are created and instantiated with values entered by the user such as home phone, wage, office phone, salary, and assistant. We chose option three from the OOTA menu to conduct a single inheritance test on `faculty4.h` and `faculty.h`. We entered the name of the class, i.e. `main_class`, the name of the method, i.e. `main_prog`, any parameters, and the two file names where the superclass `employee` and `salaried` are declared, i.e. `faculty.h` and `faculty4.h`

The test driver created by OOTA is shown in Figure 6.8. The test driver is automatically created, compiled, and linked. The test driver contains all the necessary `#include` statements of libraries of functions required for any C++ program to run. The test driver may be edited and changed if more code or libraries are needed for the compilation or test. Then it may be recompiled accordingly. The test driver also includes the name of the header file where the class definition resides.

In Figure 6.8, the header files are <faculty.h> and <faculty4.h>. An instantiation is created from the class by the statement <classname> INSTANCE1, in which INSTANCE1 is an object of the class <classname>. The <classname> is main_class and INSTANCE1 is the object of main_class. The <classname> may vary automatically (OOTA generates test drivers for all methods) or selectively (OOTA generates a test driver for one specific method) depending on the user choice. The instantiation is followed by a call INSTANCE1.<methodname> in which the <methodname> is the name of method to be tested. Again, the name of the method depends upon whether the user decides to run the automatic or the selective test driver generation. When the driver is run, the user is given the choice to append to the history file and/or log the error discovered in the history file. By having the test driver generated automatically or selectively, OOTA paves the way for better and more rigorous testing of object oriented code. The test driver will not locate the errors but can help reveal them if they exist. As shown in Figure 6.6, the error revealed was the **Salaried::bonus = salary** statement. This error causes the salary to be assigned to both local variables the salary and bonus. The statement should have been **Salaried::bonus = bonus** not **Salaried::bonus = salary**. OOTA by itself did not discover the error, however, the automated generation of test drivers and their subsequent dynamic runs aided in error discovery and increased the potential of better object oriented code testing.

```

#include <iostream.h>
#include <string.h>
class main_class
{public :void main_prog();};
class Salaried :
public employee
{
    public:
    Salaried(char *name, char *home_phone , char *office_phone, char *reports_to, float
    salary,
    float bonus_level, char *assistant);
    Salaried();
    void show_salaried(void);
    private: float salary, bonus_level;      char assistant[64];};
    Salaried::Salaried(){}
    Salaried::Salaried(char *name, char *home_phone , char *office_phone, char
    *reports_to, float salary, float bonus_level, char *assistant) : employee(name,
    home_phone, office_phone, reports_to)
    {Salaried::salary = salary;Salaried::bonus_level = salary;s
    trcpy(Salaried::assistant, assistant);} void Salaried::show_salaried(void)
    {show_employee();
    cout << "\n\n_____";
    cout << "\n\nSalary    : $" << salary << endl;
    cout << "\n\nBonus Level : " << bonus_level << endl;
    cout << "\n\nAssistant  : " << assistant << endl;
    cout << "\n\n_____";
    getch();
    }

    class hourly : public employee
    {public:hourly(char *name, char *home_phone , char *office_phone, char *reports_to,
    float wage);
    hourly();
    void show_hourly(void);
    private:      float wage;};hourly::hourly(){}
    hourly::hourly(char *name, char *home_phone , char *office_phone, char *reports_to,
    float wage) : employee(name, home_phone, office_phone, reports_to)
    { hourly::wage = wage; }
    void hourly::show_hourly(void)
    {show_employee();
    cout << "\n\nWage    : $" << wage << endl;
    }

```

Figure 6.6 Salaried, Hourly, and Temporary Classes


```

void main_class::main_prog()
{int okay, user_choice, etype; char name[64], home_phone[64], office_phone[64], reports_to[64],
assistant[64], temp1[2];
float wage, salary, bonus_level;    okay = 0;
while (okay == 0) {
cprintf("The paycheck process Menu");printf("=====\n\n");
cputs("[1]. Add a Salaried Employee\n");gotoxy(24,10);cputs("[2]. Add an Hourly Employee\n");
cputs("[3]. Add a Temporary Employee\n");gotoxy(24,12);cputs("[4]. Display information of an
Employee \n");
cputs("[5]. Exit\n"); cputs("Please Enter your choice : ");cin >> user_choice;
switch (user_choice){
case 1 : printf("\nPlease Enter Employee's Name      : ");
cin.getline(temp1,2,'\n'); cin.get(name,64);
printf("\nPlease Enter Employee's Home Phone : ");
cin.getline(temp1,2,'\n');cin.get(home_phone,64);
printf("\nPlease Enter Employee's Office Phone : ");
cin.getline(temp1,2,'\n');cin.get(office_phone,64);
printf("\nPlease Enter Employee's Boss Name : ");
cin.getline(temp1,2,'\n');cin.get(reports_to,64);
printf("\nPlease Enter Employee's Salary      : ");
scanf("%f",&salary); printf("\nPlease Enter Employee's Bonus Level : ");
scanf("%f",&bonus_level);printf("\nPlease Enter Employee's Assistant : ");
cin.getline(temp1,2,'\n');cin.get(assistant,64);
Salaried manager(name,home_phone,office_phone,reports_to,salary,bonus_level,assistant);
printf("\n\nPlease hit the Enter key to continue ");    getch(); clrscr(); break;
case 2 :
printf("\nPlease Enter Employee's Name      : ");cin.getline(temp1,2,'\n');cin.get(name,64);
printf("\nPlease Enter Employee's Home Phone : ");cin.getline(temp1,2,'\n');cin.get(home_phone,64);
printf("\nPlease Enter Employee's Office Phone : ");cin.getline(temp1,2,'\n');cin.get(office_phone,64);
printf("\nPlease Enter Employee's Boss Name : ");cin.getline(temp1,2,'\n');cin.get(reports_to,64);
printf("\nPlease Enter Employee's Wage      : ");scanf("%f",&salary);
hourly staff(name,home_phone,office_phone,reports_to,wage);
printf("\n\nPlease hit the Enter key to continue ");break;
case 3 : printf("\nPlease Enter Employee's Name      : ");cin.getline(temp1,2,'\n');cin.get(name,64);
printf("\nPlease Enter Employee's Home Phone : ");cin.getline(temp1,2,'\n');cin.get(home_phone,64);
printf("\nPlease Enter Employee's Office Phone : ");
cin.getline(temp1,2,'\n');cin.get(office_phone,64);printf("\nPlease Enter Employee's Boss Name :
");
cin.getline(temp1,2,'\n');cin.get(reports_to,64);printf("\nPlease Enter Employee's Wage      : ");
scanf("%f",&salary);
temporary tempstaff(name,home_phone,office_phone,reports_to,wage);
printf("\n\nPlease hit the Enter key to continue "); break;
case 4 :printf("\nPlease enter 1 = Salaried, 2 = Hourly, 3 = Temporary : ");scanf("%d",&etype);
switch(etype){
case 1 : manager.show_salaried(); break;
case 2 : staff.show_hourly(); break;
case 3 : tempstaff.show_temporary(); break;
default: cputs("You entered an invalid input Please try again ");    getch(); }printf("\n\nPlease

```

Figure 6.7 faculty4.h Declaration file for class faculty

```

#include <fstream.h>
#include <stdio.h>
#include <conio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <faculty.h>
#include <faculty4.h>
main()
{
main_class INSTANCE1;

INSTANCE1.main_prog();

}

```

Figure 6.8 Testprog.cpp test driver - Single Inheritance

We chose option three from the OOTA menu to conduct an multilevel inheritance test on person.h , working.h and manager.h. Figures 5.4, 6.9, 6.10, and 6.11 represent the parent classes and the subclass. We entered the number of superclasses, the name of the class, i.e. main_class, the name of the method, i.e. main_prog, any parameters, and the three file names where the superclass person, manager, and worker are declared, i.e. person.h, manager.h, and working1.h.

The file person.h contains the declaration of the class person. It contains the age and name. The method persons assigns the values of formal parameters to local parameters. The method show_person displays the name and age of the person. The file working1.h contains the declaration of the class worker and it inherits from

person. It has the local variables wage and phone. The method worker assigns the values of formal parameters to local parameters. The method show_worker displays the value of the fields age, name, and phone. The file manager.h contains the declaration of the class manager which inherits from worker and worker inherits from person. Thus, manager inherits from person and that makes this example multilevel inheritance. It has local variables as phone, wage, and age used in the method manager and assigned to the values of formal parameters.

The method show_manager displays the person's name, age, office, and phone. The class main_class is also defined in the same file manager.h. The method main_prog prompts the user to add worker information, display worker information, or exit the program. After the user enters the values, the objects are created and instantiated.

The test driver created by OOTA is shown in Figure 6.12. The test driver is automatically created, compiled, and linked. The test driver contains all the necessary #include statements of libraries of functions required for any C++ program to run. The test driver may be edited and changed if more code or libraries are needed for the compilation or test. Then it may be recompiled accordingly. The test driver also includes the name of the header file where the class definition resides. In Figure 6.12, the header files are <person.h>, <working.h> and <manager1.h>. An instantiation is created from the class by the statement <classname>

INSTANCE1, in which INSTANCE1 is an object of the class <classname>. The <classname> is main_class and INSTANCE1 is the object of main_class. The <classname> may vary automatically (OOTA generates test drivers for all methods) or selectively (OOTA generates a test driver for one specific method) depending on the user choice. The instantiation is followed by a call INSTANCE1.<methodname> in which the <methodname> is the name of method to be tested. Again, the name of the method depends upon whether the user decides to run the automatic or the selective test driver generation. When the driver is run, the user is given the choice to append to the history file and/or log the error discovered in the history file. By having the test driver generated automatically or selectively, OOTA paves the way for better and more rigorous testing of object oriented code. The test driver will not locate the errors but can help reveal them if they exist. As shown in Figure 6.11, the error revealed was the **worker(name,age,phone,wage)** statement. This error causes the invalid assignment of variables in the worker method which is worker(phone, age, name, wage). The statement should have been **worker(phone,age,name,wage)** not **worker(name,age,phone,wage)**. OOTA by itself did not discover the error, however, the automated generation of test drivers and their subsequent dynamic runs aided in error discovery and increased the potential of better object oriented code testing.

```

class person
{public:
    person(char *name, int age);
    person();
    void show_person(void);
private:char name[64];
        int age;
};
person::person()
{}

person::person(char *name, int age)
{
    strcpy(person::name, name);
    person::age = age;
}

void person::show_person(void)
{
    cout << endl << "Name : " << name << endl;
    cout << "Age : " << age << endl;
}

```

Figure 6.9 Declaration file for person class

We provided the elevator example to demonstrate the use of OOTA in object testing. We also provided the product and service classes example to demonstrate the use of OOTA in interface testing. Finally, we provided two examples of inheritance tests using OOTA. The two examples represented single and multilevel inheritance. The generation of the test drivers is an implementation of the main components. We provided examples to demonstrate that the generation of the test

drivers for all methods (inter & intra) assists the process of testing object oriented code.

```
class worker : public person
{
    public:
        worker(char *name, int age, char *phone, float wage);
        worker(); void show_worker(void);
    private:
        char phone[64];    float wage;};

worker::worker()
{}

worker::worker(char *phone, int age, char *name, float wage) :
person(name, age)
{strcpy(worker::phone, phone);
worker::wage = wage;
}

void worker::show_worker(void)
{show_person();
cout << "Phone : " << phone <<
endl;
}
```

Figure 6.10 Declaration file for worker class

```

#include <iostream.h>
#include <string.h>
class main_class {
    public: void main_prog();
};
class manager: public worker
{
    public:
        manager(char *name, int age, char *phone, float wage, char
*office);
        manager();
        void show_manager(void);
    private:
        char office[64];
};
manager::manager(){}
manager::manager(char *name, int age, char *phone, float wage, char *office) :
worker(name, age, phone, wage)
{
    strcpy(manager::office, office);}
void manager::show_manager(void)
{show_worker();    cout << "\n\Office    : " << office << endl;}
void main_class::main_prog()
{int okay, user_choice, etype, age; char name[64], phone[64], office[64],
temp1[2];float wage; okay = 0;
while (okay == 0) {cprintf ("The Worker's Directory Menu");
cprintf ("===== \n\n");
cputs("[1]. Add a worker's information\n");gotoxy(24,10);cputs("[2]. Display
worker's Information\n");gotoxy(24,11);cputs("[3].Exit\n");
cputs("Please Enter your choice : ");cin >> user_choice;
switch (user_choice){
case 1 :printf("\nPlease Enter Employee Name                :
");cin.getline(temp1,2,'\n');cin.get(name,64);
printf("\nPlease Enter Employee Phone Number    : ");
cin.getline(temp1,2,'\n');cin.get(phone,64);
printf("\nPlease Enter Age of Employee          : ");
scanf("%d",&age);printf("\nPlease Enter Employee's Wage      : ");
scanf("%f",&wage);printf("\nPlease Enter Employee's Office location : ");
cin.getline(temp1,2,'\n');cin.get(office,64);
manager boss(name,age,phone,wage,office);
printf("\n\nPlease hit the Enter key to continue ");getch(); break;
case 2 :clrscr();boss.show_manager(); getch(); break;
case 3 : okay = 1; break;
default: cputs("You entered an invalid input, Please try again..");getch();

```

Figure 6.11 Declaration file for manager class

```
#include <fstream.h>
#include <stdio.h>
#include <conio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <person.h>
#include <working.h>
#include <manager1.h>
main()
{
main_class INSTANCE1;
INSTANCE1.main_prog();
}
```

Figure 6.12 Testprog.cpp test driver – multiple inheritance

CHAPTER 7. SUMMARY

The objective of this work was to improve the process of testing object-oriented systems. We defined a framework that generated a testing assistant called OOTA. The results of this research show that an assistance process facilitates rigorous testing of object-oriented code. The assistance makes the testing process more efficient and productive. The application of the test drivers generated from the testing assistant enhances the likelihood of thorough testing. Also, the testing assistant can be reused to test other object-oriented systems.

This research was divided into five major sections corresponding to chapters 2 to 6 in this document: related work, fundamental components of an object oriented testing assistant, functionality of the testing assistant, architecture of the testing assistant, and testing assistant case studies.

Chapter 2 presented work related to this research. It included four major categories of research conducted on testing object oriented code: 1) single testing strategy, 2) testing frameworks with theories, 3) implementations of testing strategies, and 4) assessment of testing strategies. A comparison of the related work was given in table 2.1.

Chapter 3 described the motivation for pursuing the automated assistance process to improve method testing. Two testing techniques were presented as the major components of the testing assistant. The first component was Object Level

Test (OLT). It included two tests: Object Test (OT) and Interface Test (IT). In OT all methods are tested in the object. In IT, the methods involved in the interface between objects are tested. The second component was Test for Inheritance within Classes (TIC). It involved testing the inherited methods in all forms: single, multiple, and multilevel. Both components were presented with a detailed description and examples to support the need for these two components.

Chapter 4 was devoted to the functionality of the testing assistant. The production of the testing assistant was presented in the form of algorithms. The algorithms implement the test described in chapter 3. Three different algorithms were presented, one for each of the tests OT, IT, and TIC. The algorithms generate the test driver automatically (for all methods) or selectively (for one specific method). Two additional computation algorithms were also presented which map to the generation algorithms. The number of test drivers computed in the computation algorithm should match the number of test drivers generated by the generation algorithms.

Chapter 5 described the architecture of the testing assistant. It included analysis and description of the data structure, test data set, implementation, and features of the testing assistant OOTA. The system layout was described in detail in figure 5.1. The technical runs were described by showing sample of screen runs and output from OOTA.

Finally, chapter 6 included case studies of the testing assistant. This chapter illustrated the benefits of OOTA by the use of three case studies. For each case study, we included a description of the case study and a description of the results obtained from OOTA. The testing techniques OLT and TIC were demonstrated by the implementation of OOTA with the case studies.

7.1 Conclusions

The conclusion of this research is that the automated testing assistant and test driver generation process increase the potential for better method testing. In particular, this research shows that the automated assistance process facilitates the testing process of object oriented code. The automated assistant has the following features:

- 1) OOTA creates, compiles, and runs the test drivers created internally within OOTA.
- 2) OOTA maintains a history file that can be revisited at any time as a reference for future occurrence of errors. Revisiting the history file assists in reducing unnecessary testing and preventing errors from recurring.
- 3) OOTA inheritance test (TIC) is done for the single/multiple inheritance and for the multilevel inheritance.
- 4) OOTA generated test drivers may be saved and reused later to test other object oriented systems.

- 5) OOTA generated test drivers can be selective (generated by OOTA for one specific method) or automatic (generated by OOTA for all methods).
- 6) OOTA improved the potential of thorough method testing.
- 7) OOTA is easy to use and menu driven.
- 8) OOTA has a standardized class structure.
- 9) OOTA provides a framework that helps ensure appropriate components and interactions are tested by generating code segments that drive the testing process.

The assistance process is practical since the generated test drivers are readily available. This assistance will ensure that methods are inter and intra tested.

It adds efficiency to the testing process by acting as a tool that helps to conduct rigorous testing. It enhances testing by making the testing process more flexible. The assistance process enhances the chances of error discovery. Since the automated assistance encourages the use of the generated test drivers, the more tests that are run, the greater the chances of errors discovery. The assistance will not locate or discover the error but it facilitates the process of error discovery. During each of the eighty eight dynamic runs done with OOTA, the errors detected were stored in a history file. While OOTA did not discover these errors, the test drivers generated by OOTA helped to find the errors.

The process that generated the automated testing assistant, OOTA, involved the following steps: 1) defining the components of the testing assistant,

2) implementing the component of the testing assistant, 3) exercising the testing assistant, and 4) analyzing the results. We were successful in implementing OOTA as an object oriented automated test driver generation tool. As demonstrated in chapter 6, experimentation with OOTA provided positive results. The implementation of the object test, interface test, inheritance test, and the availability of the automatic and/or the selective generation of test drivers facilitates the testing of all methods.

OOTA was developed with the intention of 1) transforming the steps of a testing framework into procedures, 2) reusing the tool to test other systems, and 3) analyzing results for future reference. The primary motivation behind OOTA was to improve the likelihood that methods will be thoroughly tested. We defined algorithms to help ensure that all methods are inter and intra tested.

The testing techniques (OLT & TIC) were the focus of the devised automated test driver generation process. Method coverage was the main goal of OT, and methods interface was the main goal of IT. With TIC, we tested all types of inheritance: single, multiple, and multilevel. The algorithm OLT and the algorithm TIC were implemented in OOTA. OOTA derives test drivers for OLT and TIC.

The testing of object-oriented programs is still at an immature stage. This research was devoted to this problem. It focused on defining a testing assistant in the form of an automated object oriented driver generation process. The testing

assistant allows the user/tester to base the testing on the areas of the system that cause errors.

7.2 Future Work

This research enhances object oriented testing. However, there are issues that can be explored in future research:

- 1) Expand the assistant to handle more complex code.
- 2) Make the assistant process programs in other OO languages.
- 3) Expand this research to develop new testing techniques.

Improvements to the automated testing assistant can be made to handle more complex code. OOTA could work at the class level instead of just at the object level, as it does currently. Additionally, we can expand the automated testing assistant to handle any object oriented language. The user may then be able to select the language of his/her choice to generate the test driver written for that language.

There is also room for improvement in the algorithms which are the components of the assistant. More algorithms can be written to expand the level of testing to include the class level. We could apply OOTA to more data sets and reuse it in other object oriented code to improve the efficiency and scope of the assistant.

REFERENCES

- [Arnold94] Arnold, T. and W. Fuson, "Testing "In A Perfect World"", Communication of the ACM, September 1994, Vol. 37, No. 9, pp. 79-86.
- [Askit92] Askit, M. and L. Bergmans, "Obstacles in Object-Oriented Software Development", OOPSLA'92, pp.341-358.
- [Beizer90] Beizer, B., Software Testing Techniques, Van Nostrand Reinhold, 1990.
- [Berard96] Berard, E. V., "Bringing Testing Into The Fold", IEEE Software, May 1996, pp. 91-95
- [Bergadano96] Bergadano, F., and D. Gunetti, "Testing by Means of Inductive Program Learning", ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 2, April 1996, pp. 119-145
- [Binder94] Binder, R., "Design for testability in Object-Oriented Systems", Communication of the ACM, September 1994, Vol. 37, No.9, pp. 87-101
- [Chaar93] Chaar, J. K., M. J. Halliday, I. S. Bhandari, and R. Chillarege, "In-Process Evaluation for Software Inspection and Test", IEEE Transactions on Software Engineering, Vol. 19, No. 11, November, 1993, pp. 1065-1069.
- [Cheatham90] Cheatham, T. and L. Mellinger, "Testing Object-Oriented Software Systems", In Proceedings of the 1990 ACM Eighteenth Annual Computer Science Conference Proceedings, pp. 161-165, ACM, February 1990.
- [Chidamber91] Chidamber, S. R., and C. F. Kemerer, "Towards a Metric Suite for Object Design", OOPSLA'91, pp. 197-211.
- [Collefello90] Collefello, J. , T. Fisher, and M. Rees M., "A Testing Methodology Framework", IEEE Computer Software and Application Conference, 1990, (14th) IEEE Annual International, pp. 577-586.
- [DeMillo87] DeMillo, McCracken, Martin, and Passafiume, Software Testing and Evaluation, 1987, Benjamin \ Cummings.

[DeMillo91] DeMillo, R., "Progress Toward Automated Software Testing", IEEE Software Engineering 1991 (13th) International Conference on (ICSE-13), pp. 180-183

[DeMillo95] DeMillo, R., A. O. Mathur, and W. E. Wong, "Some Critical Remarks on a Hierarchy of Fault-Detecting Abilities of Test Methods", IEEE Transactions on Software Engineering, Vol. 21, No. 10, October 1995, pp. 858-861

[Devanbu96] Devanbu, P. T., D. S. Rosenblum, and A. Wolf, "Generating Testing and Analysis tools with Aria", ACM Transaction on Software Engineering and Methodology, Vol. 5, No. 1, Jan 1996, pp. 42-62

[Doan95] Doan, D. K., N. W. Paton, and A. Kilgour, "Design and User Testing of A Multi-Paradigm Query Interface to An Object-Oriented Database", SIGMOD Record, Vol. 24, No. 3, September 1995, pp. 12-17

[D'Souza94] D'Souza, R. J., and R. LeBlanc, Jr., "Class Testing by Examining Pointers", Journal of Object Oriented Programming, July - August 1994, pp. 33-39.

[Fayad93] Fayad, M., L. J. Hawn, M. A. Roberts, and J. R. Klatt, "Using the Shaler-Mellor Object-Oriented Analysis Method", IEEE Software, March 1993, pp. 43- 2.

[Gannon81] Gannon, J. , P. McMullin, and R. Hamlet, "Data Abstraction Implementation, Specification and Testing", ACM Trans. Program. Lang. Syst., 3(3), pp. 211-223, July 1981

[Harold94] Harold, M. J., J. D. McGregor, and K. J. Fitzpatrick, "Incremental Testing of Object-Oriented Class Structure", In Proceedings of the 1992 ACM Twentieth Annual Computer Science Conference Proceedings, pp. 68-79, ACM, February 1992.

[Hekmatpour90] Hekmatpour, S., C++ A Guide for C Programmers, Prentice Hall, 1990.

[Hoffman89] Hoffman, D., "A Case in Module Testing", In Proceedings of the Conference on Software Maintenance 1989, IEEE Computer Society, Washington D.C., 1989, pp. 100-105

[Holub93] Holub, A., "The Power of Inheritance", BYTE, May 1993, pp. 221-226.

[Howden78] Howden, W., "Functional Program Testing", I E E Trans. Software Eng. SE-6(2), pp. 162-169

[Howden95] Howden, W., and Y. Huang, "Software Trustability Analysis, ACM Transactions on Software Engineering and Methodology, Vol. 4, No. 1, January 1995, pp. 36-64

[Hughes96] Hughes, M., and D. Stotts, "Daistish: Systematic Algebraic Testing of OO Programs in the Presence of Side-effects", Software Engineering Notes Vol. 21, No. 3, May 1996, Proceedings of 1996 International Symposium on Software Testing and Analysis (ISSTA), Jan 8 - 10, 1996, pp. 53-61

[Jamsa95] Jamsa, K., Success With C⁺⁺, Boyd & Fraser, 1995.

[Jorgensen94] Jorgensen, P. C., and C. Erickson, "Object-Oriented Integration Testing", Communication of The ACM, September 1994, Vol. 37, No. 9, pp. 30-38.

[Khan95] Khan, E. H., M. Al-A'ali, M. R. Girgis, "Object-Oriented Programming for Structural Procedural Programmers, IEEE Computer, October 1995, pp. 48-57

[Korel96] Korel, B., and A. M. Al-Yami, "Assertion-Oriented Test Data Generation", 1996 IEEE 18th International Conference on Software Engineering, pp. 71-80

[Kung95a] Kung, D., J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "A Test Strategy for Object-Oriented Programs", 1995 IEEE 19th Annual International Computer Software and Application Conference, pp. 239-243

[Kung95b] Kung, D., J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "Object State Testing For Object-Oriented Programs", 1995 IEEE 19th Annual International Computer Software and Application Conference, pp. 232-238

[Lieberherr93] Lieberherr, K. and C. Xiao, "Object-Oriented Software Evolution", IEEE Transactions on Software Engineering, Vol 19, No. 4, April 1993, pp. 313-342.

[Marick95] Marick, B., The Craft of Software Testing, Prentice Hall, 1995.

[Maring96] Maring, B., "Object-Oriented Development of Large Applications", IEEE Software, May 1996, pp. 33-40

[McGregor94] McGregor, J. and T. Korson, "Integrated Object-Oriented Testing and Development Processes", *Communication of the ACM*, September 1994, Vol. 37, No. 9, pp. 59-77.

[Murphy92] Murphy, G. L., "Towards a Testing Methodology for Object - Oriented Systems", *OOPSAL'92*, Addendum to the Proceedings, 5-10 October, pp. 139-140

[Murphy94] Murphy, G., Townsend P., and P. Wong, "Experiences with Cluster and Class Testing", *Communication of the ACM*, September 1994, Vol. 37, No. 9, pp. 39-47.

[Myers79] Myers, G. J., *The Art of Software Testing*, 1979, Wiley, New York.

[Offutt96] Offutt, A. J., and J. H. Hayes, "A Semantic Model of Program Faults", *Software Engineering Notes* Vol. 21, No. 3, May 1996, *Proceedings of 1996 International Symposium on Software Testing and Analysis (ISSTA)*, Jan 8 - 10, 1996, pp. 195-199

[Offutt98] Offutt, A. J "Software Testing: From Theory to Practice", *IEEE AES Systems Magazine*, March 1998, Vol. 13, No. 3, pp. 195-199

[Oualline95] Oualline, S., *Practical C++ Programming*, O'Reilly & Associates, INC., 1995.

[Parrish93] Parrish, A., D. Cordes, and R. Borie, "Developmental Testing of Abstract Data Type", *COMPSAC / IEEE Computer Society's International Computer Software & Application Conference Proceedings*, 1993, pp. 49-55.

[Pfleeger98] Shari, Lawrence, and Pfleeger, S. L., *Software Engineering*, Prentice hall Publishing Company, 1998.

[Poston94] Poston, R. M., "Automated Testing from Object Models", *Communication of the ACM*, September 1994, Vol. 37, No. 9, pp. 48-58.

[Pressman97] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, 4th edition, McGraw-Hill, 1997

[Rettig91] Rettig, M., "Testing made Palatable", *Communication of the ACM*, May 1991, Vol. 34, No. 5, pp. 25-29.

- [Sametinger90] Sametinger, J., "A Tool for the Maintenance of C⁺⁺ Programs, IEEE Software Maintenance 1990 (6th) Conference on (CSM 90), pp. 54-59.
- [Schildt94] Schildt, H., C⁺⁺ From The Ground Up, McGraw-Hill, 1994.
- [Singh93] Singh, G. B., "Single Versus Multiple Inheritance in Object Oriented Programming", In Proceedings of the 1990 ACM Twenty First Annual Computer Science Conference Proceedings, pp. 34-43, ACM, February 1993.
- [Stevens93] Stevens, A., Teach Yourself C⁺⁺, MIS Press, 1993.
- [Voas95] Voas, J. M., and K. W. Miller, "Software Testability: The New Verification", IEEE Software, May 1995, pp.17-28
- [White87] White, L. J., Software Testing and Verification, 1987, Academic Press.
- [Wong95] Wong, W.E., J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness", 1995 IEEE 17th International Conference on Software Engineering, pp. 41-50
- [Xiaoping93] Xiaoping, J., " Model-Based Formal Specification Directed Testing of Abstract Data Types", C O M P S A C / I E E E Computer Society's International Computer Software & Application Conference Proceedings, 1993, pp. 577-586.

VITA

Ihssan S. Alkadi is a native of Lebanon. He acquired his Bachelor of Science degree from Southeastern Louisiana University, Hammond, in May 1985. He acquired his Master of Science degree from Louisiana State University, Baton Rouge, in May 1992. His profession is a full time faculty at S.L.U since fall 1992 until present. He will receive the degree of Doctor of Philosophy in August 1999 from L.S.U.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Ihssan Salim Alkadi

Major Field: Computer Science

Title of Dissertation: Enhancing the Process of Testing Object-Oriented Systems

Approved:

Morris L. Carter
Major Professor and Chairman

Sam M. Larkin
Dean of the Graduate School

EXAMINING COMMITTEE:

Leifloch

J B Jones

1. def y

Wangjun Wang

_____

_____

_____

© 2006 The Authors
Journal compilation © 2006 Blackwell Publishing Ltd

Date of Examination:

May 12, 1999