UML Design Based Testing

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
In
Software Engineering

Thapar University, Patiala

By:
Gurpreet Singh
(80631006)

Under the supervision of:
Mr. Rajesh Kumar Bhatia
Assistant Professor
CSED, TU, Patiala

JUNE 2008

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Software Engineering"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Mr. Rajesh Kumar Bhatia and refers other researcher's works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

(**Gurpreet Singh)**

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(**Mr. Rajesh Kumar Bhatia)**
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

(Dr. (Mrs.) Seema Bawa)
Professor & Head
Computer Science & Engineering. Department
Thapar University
Patiala

(Dr. R.K.Sharma)
Dean (Academic Affairs)
Thapar University,
Patiala.

Acknowledgment

I wish to express my deep gratitude to Mr. Rajesh K. Bhatia, Assistant Professor and computer Science & engineering department, TU, Patiala for providing his uncanny guidance and support throughout the thesis.

I am thankful to Dr. Seema Bawa, Head, Computer Science & Engineering Department, TU, Patiala for the motivation and inspiration that triggered me for the thesis work.
I would also like to thank all the staff members and my co-students who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of the thesis.

Last but not the least, I express my heartfelt thanks to my parents, my brothers, other members of staff, my friends and well-wishers for co-operation, which they were always ready to extend.

Gurpreet Singh
80631006
M.E.(Software Engineering)-2nd year
Computer Science & Engineering Department
Thapar University
Patiala -147004

Abstract

Testing is very important part of software development. Almost 80% software fail because of not testing properly. Testing is performed by different types of strategies. Generally testing is performed on code, but if the software can be tested in the earlier phases then most of the errors can be eliminated and can be stopped from propagating to next phase. Thus there is a need to explore testing possibilities in earlier phases.

The proposed work presents a novel design based testing approach that can fix errors in initial phase. To perform design based testing, we need a language that can deal with the design efficiently i.e. Unified Modeling Language (UML). UML consists of different designs that are used to specify the static and dynamic behavior of the software. UML diagrams are used for modeling object-oriented systems, not on structured systems. In the proposed system testing is performed on two diagrams i.e. class diagram and sequence diagram. Proposed system focuses on these two diagrams and with the help of these diagrams we can find out optimal solution and perform better testing than previous approaches.

- Class diagram based testing solves the problem of static test cases with automated method.
- Sequence diagram based testing solves the problem of dynamic test cases.
- Further with help of UML we can perform regression testing i.e. finding affect of change in design.

The proposed work is divided into several modules that generate automated test cases for design. The modules are class analyzer, function analyzer, static test cases generator, sequence diagram based testing, dynamic test cases generator and regression testing module.

Table of Contents

Certificate.......... i
Acknowledgement.......... ii
Abstract.......... iii
Table of Contents.......... iv
List of Figures.......... vii

Chapter 1 Introduction……1
1.1 Object-Oriented Testing……2
1.2 Object-Oriented Testing ……..2
1.3 Motivation & Objective…...3
1.4 Organization of Thesis…..4
Chapter 2 Background Information And Literature Review…...5
2.1 History of UML……5
2.2 What is UML?…...7
2.2.1 Goals of UML…..7
2.2.2 Why Use UML?…...7
2.3 Types of UML Diagrams…..8
2.4 Class Diagram…...9
2.4.1 When to Use: Class Diagrams…...11
2.5 Sequence Diagram…..11
2.6 Testing overview……13
2.6.1 Black Box Testing…..13
2.6.2 Advantages of Black Box Testing……14
2.6.3 Disadvantages of Black Box Testing……14
2.6.4 Testing Strategies/Techniques…...14
2.7 White Box Testing…..15
2.8 Review of State of Art……16

2.8.1 UML Class Diagram Related Work…………………………………...16
2.8.1.1 UML Class Diagram Based Testing Using Verification……..16
2.8.2 UML Sequence Diagram Related Work……………………………18
2.8.2.1 UML Sequence Diagram Based Testing Using Slicing………18
2.8.2.2 Sequence Diagram Testing Using Verification………………19
2.8.2.3 UML Sequence Diagrams And State Diagrams……………...20
Chapter 3 Problem Statement……………………………………………………...22
3.1 Gaps in Existing Work……………………………………………………22
3.2 Problem Formulation……………………………………………………..23
Chapter 4 Proposed System And Implementation……………………………………25
4.1 Introduction…………………………………………………………………25
4.1.1 Class Diagram……………………………………………………………..25
4.1.2 Sequence Diagram………………………………………………………...25
4.1.3 Combination of Class and Sequence Diagram…………………………25
4.2 Brief Overview About System…………………………………………26
4.3 UML Class Diagram Based Testing………………………………………...26
4.3.1 Class Analyzer………………………………………………………26
4.3.2 Details of Class Analyzer………………………………………28
4.3.3 Class Path Mechanism……………………………………………28
4.4 Function Analyzer…………………………………………………………...28
4.4.1 Class Based Test Cases…………………………………………30
4.4.2 Details of Function Analyzer…………………………………...31
4.4.3 Function Analyzer Mechanism…………………………………32
4.5 Sequence Diagram And Dynamic Test Cases……………………………32
4.5.1 Generation of Dynamic Test Cases……………………………..33
4.6 UML Design Based Regression Testing…………………………………35
4.7 Working of UML Based Testing Tool……………………………………38
4.7.1 Working steps……………………………………………………..38
4.8 Working of Regression Testing Tool……………………………………..39
4.8.1 Working Steps……………………………………………………..39
Chapter 5 Conclusions & Future Work………………………………………………41

5.1 Conclusions……………………………………………………….41
5.2 Future Work………………………………………………………41
References…………………………………………………………………………….43
List of Papers/Publications……………………………………………………………45

List of Figures

Fig 2.1 Class Diagram …………………………………………………………………9
Fig 2.2 Relationship Between Diagrams…………………………………………...10
Fig 2.3 Generalization………………………………………………………...10
Fig 2.4 Sequence Diagram……………………………………………………11
Fig 2.5 Message Sequence (a)……………………………………………..12
Fig 2.5 Message Sequence (b)……………………………………………..12
Fig 2.7 Message Sequence (c)……………………………………………...13
Fig 4.1Class Analyzer….…………………………………………………..26
Fig 4.2 Dependency Flow Graph……………………………………………..27
Fig 4.3 UML Class Diagram…………………………………………………28
Fig 4.4 Class Path Graph……………………………………………………...28
Fig 4.5 Function Analyzer……………………………………………………29
Fig 4.6 Class/Function Flow Graph…………………………………………..29
Fig 4.7 Static Test Cases……………………………………………………...30
Fig 4.8 UML Class Diagram…………………………………………………31
Fig 4.9 Class and Function Path………………………………………………32
Fig 4.10 Sequence Testing Module…………………………………………...33
Fig 4.11(a) Dynamic Test Cases……………………………………………...33
Fig 4.11(b) Dynamic Test Cases……………………………………………...34
Fig 4.12 Dynamic test cases & sequence diagram……………………………34
Fig. 4.13 Regression Testing…………………………………………………35
Fig 4.14 Overall Working of System…………………………………………38
Fig 4.15 Regression Testing Overview……………………………………….39

Chapter 1
Introduction

IEEE definition of software testing is executing the program with intent of finding errors. Software testing is an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate. Quality is not an absolute term; it is value to some person. With that in mind, testing can never completely establish the correctness of arbitrary computer software; testing furnishes a criticism or comparison that compares the state and behaviour of the product against a specification.

A common practice of software testing is performed by an independent group of testers after the functionality is developed before it is shipped to the customer. This practice often results in the testing phase being used as buffer to compensate for project delays, thereby compromising the time devoted to testing. Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.

Software testing process can produce several artifacts. The software testing document, which consists of event, action, input, output, expected result, and actual result. Clinically defined a test case is an input and an expected result. This can be as pragmatic as 'for condition x your derived result is y', whereas other test cases described in more detail the input scenario and what results might be expected. It can occasionally be a series of steps (but often steps are contained in a separate test procedure that can be exercised against multiple test cases, as a matter of economy) but with one expected result or expected outcome. The optional fields are a test case ID, test step or order of execution number, related requirement(s), depth, test category, author, and check boxes for whether the test is automatable and has been automated. Larger test cases may also contain prerequisite states or steps, and descriptions. A test case should also contain a place for the actual result. These steps can be stored in a word processor document, spreadsheet, database, or other common repository. In a database system, you may also be able to see past test results and who generated the results and the system configuration used to

generate those results. These past results would usually be stored in a separate table.

1.1 Object-Oriented Testing

Object-Oriented testing basically focus on classes and objects. These testing techniques focus on classes and objects. Features such as class inheritance and interfaces support polymorphism in which code manipulates objects without their exact class being known. Testers must ensure that the code works no matter what the exact class of such objects might be. Features that support data hiding complicate testing because operations must be added to a class interface by the *developer* just to support testing.categories of object oriented testing:

• Model testing
• Class testing instead of unit testing
• Class interaction testing instead of integration testing
• System and subsystem testing
• Acceptance testing
• Self-testing

1.2 UML Design Based Testing

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is very important for developing object oriented software. The UML uses mostly graphical notations to express the design of software projects. Use of UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

Testing generally divided into two kinds:

- Verification
- Validation

So design based testing mainly focus on verification testing. The promising alternative for testing object-oriented softwares is use of UML based designs. UML based designs give detailed specifications of expected functionality. Why we

need verification testing on the design? Because a good design leads to appropriate code, but a poor design of software results in poor coding. The poor software design may suggest inappropriate test cases, which further effects the end product. So appropriate and good quality design is essential for producing testable code. UML diagrams may be used for generating test cases automatically. As class diagram elaborate the static behavior of software, main focuses is on classes and relationships like generalization and dependency. Sequence diagram elaborate the dynamic behavior of software. In sequence diagram we focus on message calling between object and class with different relationships like calling message, calling return etc.

1.3 Motivation & Objective

Testing is very important part of software development process. It takes half of the time of the software development. Testing is performed on the requirements, code and design with different methodologies. But still failure rate of the software project is about 80%. The reason for failure is because the testing performed on the code is not sufficient to detect various types of errors in the software.

Generally practitioners consider code testing as most important. As the behavior of the executable code can be tested and practitioners consider only faults in code as software failure. So testing only code and ignoring all other artifacts may result in catastrophic type of failure. Design phase is very important part because whole coding part depend on this phase. But generally testing of software design is avoided or done partially. Because design is not executable part. So testing of a design becomes very cumbersome. To solve that generally unified modeling language (UML) is used. UML deals with object oriented design method. UML has different digrams that specify all design behaviors like static and dynamic behavior. In present work UML design based automatic testing is proposed with following objectives:

- To explore existing desgin based testing techniques.
- To investigate UML based object oriented design testing.
- To develop a UML based automated design testing technique.

1.4 Organization of Thesis

This report is divided into six chapters.Chapter 1 gives brief introduction about testing, object oriented testing, UML design based testing. Objectives of the proposed work are also considered here. Chapter 2 gives brief intoduction to UML, its diagram, testing tactics and review of literature on UML based testing. Chapters 3 specify problem statement in old system and what is new in proposed work. Chapter 4 gives information of proposed system with some snapshots of UML design based tool.Chapter 5 focus on conclusion and future scope.

Chapter 2
Background Information And Literature Review

The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing Object Oriented software and the software development process. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, and load balancing and fault tolerance.

2.1 History of UML

Identifiable object-oriented modeling languages began to appear between mid-1970 and the late 1980s as various methodologists experimented with different approaches to object-oriented analysis and design [19]. The number of identified modeling languages increased from less than 10 to more than 50 during the period between 1989-1994. Many users of OO methods had trouble finding complete satisfaction in any one modeling language. By the mid-1990s, new iterations of these methods began to appear and these methods began to incorporate each other's techniques, and a few clearly prominent methods emerged.

The development of UML began in late 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. In the Fall of 1995, Ivar Jacobson and his Objectory company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method.

As the primary authors of the Booch, OMT, and OOSE methods, Grady Booch, Jim Rumbaugh, and Ivar Jacobson were motivated to create a unified modeling language for three reasons. First, these methods were already evolving toward each other independently. It made sense to continue that evolution together rather than apart, eliminating the potential for any unnecessary and gratuitous differences that would further confuse users. Second, by unifying the semantics and notation, they could bring some stability to the object-oriented marketplace,

allowing projects to settle on one mature modeling language and letting tool builders focus on delivering more useful features. Third, they expected that their collaboration would yield improvements in all three earlier methods, helping them to capture lessons learned and to address problems that none of their methods previously handled well.

The efforts of Booch, Rumbaugh, and Jacobson resulted in the release of the UML 0.9 and 0.91 documents in June and October of 1996. During 1996, the UML authors invited and received feedback from the general community. They incorporated this feedback, but it was clear that additional focused attention was still required.

While Rational was bringing UML together, efforts were being made on achieving the broader goal of an industry standard modeling language. In early 1995, Ivar Jacobson (then Chief Technology Officer of Objectory) and Richard Soley (then Chief Technology Officer of OMG) decided to push harder to achieve standardization in the methods marketplace. In June 1995, an OMG-hosted meeting of all major methodologists (or their representatives) resulted in the first worldwide agreement to seek methodology standards, under the aegis of the OMG process.

During 1996, it became clear that several organizations saw UML as strategic to their business. A Request for Proposal (RFP) issued by the Object Management Group (OMG) provided the catalyst for these organizations to join forces around producing a joint RFP response. Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML 1.0 definition. Those contributing most to the UML 1.0 definition included: Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys. This collaboration produced UML 1.0, a modeling language that was well defined, expressive, powerful, and generally applicable. This was submitted to the OMG in January 1997 as an initial RFP response.

In January 1997 IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam also submitted separate RFP responses to the OMG.

These companies joined the UML partners to contribute their ideas, and together the partners produced the revised UML 1.1 response. The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners. It was submitted to the OMG for their consideration and adopted in the fall of 1997.

2.2 What is UML

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems [19]. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing Object Oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

2.2.1 Goals of UML

The primary goals in the design of the UML were:

- Provide users with a ready-to-use, expressive visual modeling language so they provide extensibility and specialization mechanisms to extend the core concepts.
- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the OO tools market.
- Support higher-level development concepts such as collaborations, frameworks, patterns and components.

2.2.2 Why Use UML?

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component

technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, and load balancing and fault tolerance. Additionally, the development for the World Wide Web, while making some things simpler, has exacerbated these architectural problems. The Unified Modeling Language (UML) was designed to respond to these needs.

2.3 Types of UML Diagrams

Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. UML diagrams commonly created in visual modeling tools include:

- Use Case Diagram displays the relationship among actors and use cases.
- Class Diagram models class structure and contents using design elements such as classes, packages and objects. It also displays relationships such as containment, inheritance, associations and others.
- Sequence Diagram displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).
- Collaboration Diagram displays an interaction organized around the objects and their links to one another. Numbers are used to show the sequence of messages.
- State Diagram Displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli, together with its responses and actions.
- Activity Diagram displays a special state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing.
- Component Diagram displays the high level packaged structure of the code itself. Dependencies among components are shown, including source code

components, binary code components, and executable components. Some components exist at compile time, at link time, at run times well as at more than one time.

- Deployment Diagram displays the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units.

Proposed work mainly focus on class diagram and sequence diagram, so we discuss these two diagrams in detail below.

2.4 Class Diagram

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects [19]. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design. This example is only meant as an introduction to the UML and class diagrams. Classes are composed of three things: a name, attributes, and operations. Below is an example of a class.

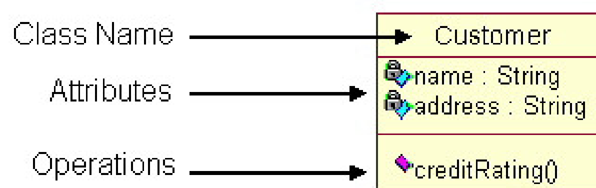


Fig 2.1 Class Diagram

Class diagrams also display relationships such as containment, inheritance, associations and others. Below is an example of an associative relationship:

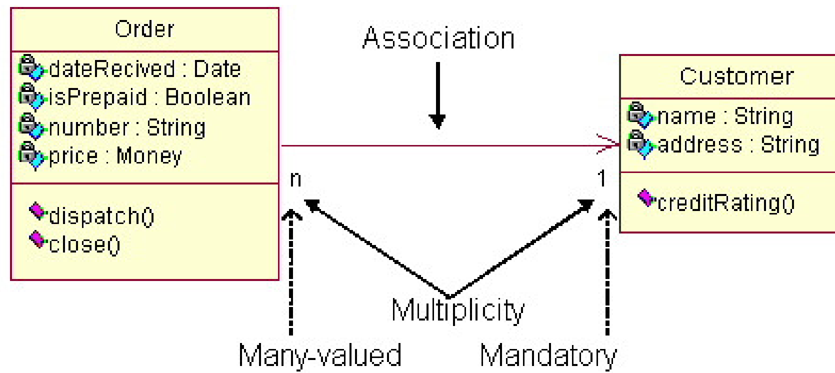


Fig 2.2 Relationship Between Diagrams

The association relationship is the most common relationship in a class diagram. The association shows the relationship between instances of classes. For example, the class Order is associated with the class Customer. The multiplicity of the association denotes the number of objects that can participate in then relationship. For example, an Order object can be associated to only one customer, but a customer can be associated to many orders. Another common relationship in class diagrams is a generalization. A generalization is used when two classes are similar, but have some differences. Look at the generalization below:

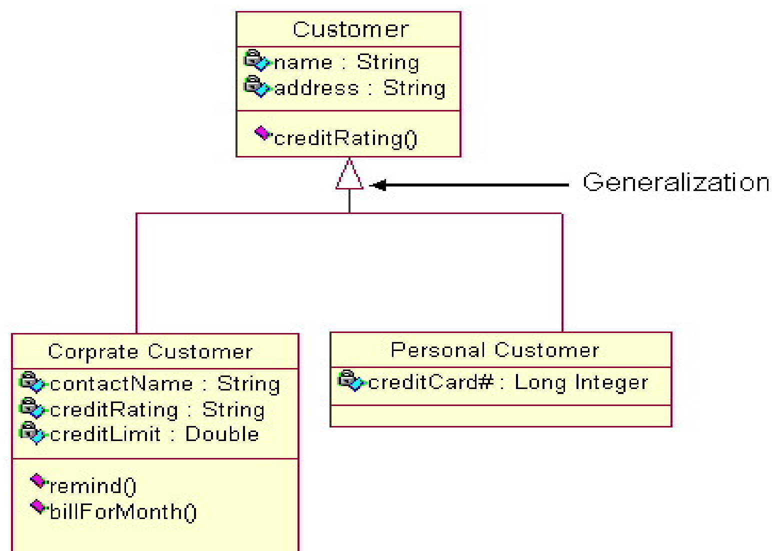


Fig 2.3 Generalization

In this example the classes Corporate Customer and Personal Customer have some similarities such as name and address, but each class has some of its own attributes and operations. The class Customer is a general form of both the Corporate Customer and Personal Customer classes. This allows the designers to just use the Customer class for modules and do not require in-depth representation of each type of customer.

2.4.1 When to Use: Class Diagrams

Class diagrams are used in nearly all Object Oriented software designs. Use them to describe the Classes of the system and their relationships to each other.

2.5 Sequence Diagram

Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass [19]. The diagrams are read left to right and descending. The example below shows an object of class 1 start the behavior by sending a message to an object of class 2. Messages pass between the different objects until the object of class 1 receives the final message.

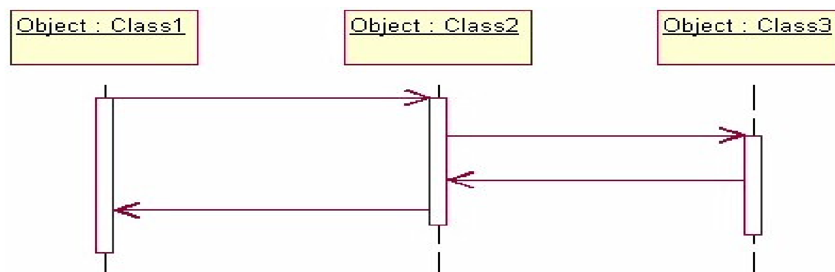


Fig 2.4 Sequence Diagram

Below is a slightly more complex example. The light blue vertical rectangles the objects activation while the green vertical dashed lines represent the life of the object. The green vertical rectangles represent when a particular object has control. The represents when the object is destroyed. This diagrams also shows conditions for messages to be sent to other object. The condition is listed between brackets next to the message. For example, a [condition] has to be met before the object of class 2 can send a message() to the object of class 3.

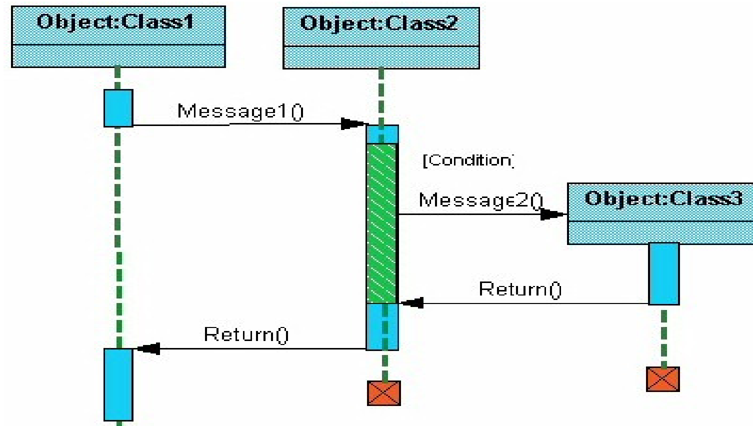


Fig 2.5 Message Sequence (a)

The next diagram shows the beginning of a sequence diagram for placing an order. The object an Order Entry Window is created and sends a message to an Order object to prepare the order. Notice the names of the objects are followed by a colon. The names of the classes the objects belong to do not have to be listed. However the colon is required to denote that it is the name of an object following the objectName: className-naming system.

Next the Order object checks to see if the item is in stock and if the [InStock] condition is met it sends a message to create an new Delivery Item object.

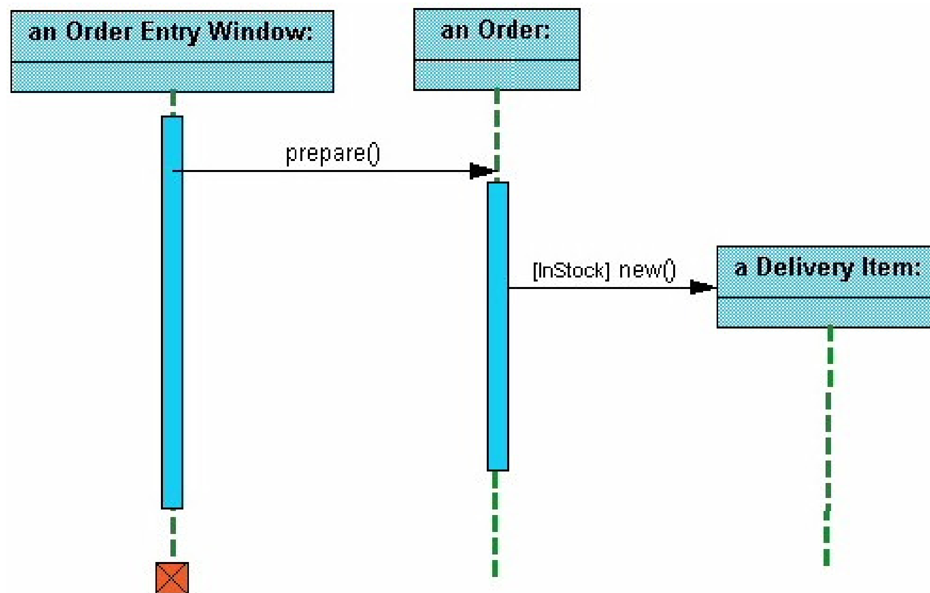


Fig 2.5 Message Sequence (b)

The next diagrams add another conditional message to the Order object. If the item is [OutOfStock] it sends a message back to the Order Entry Window object stating that the object is out of stack.

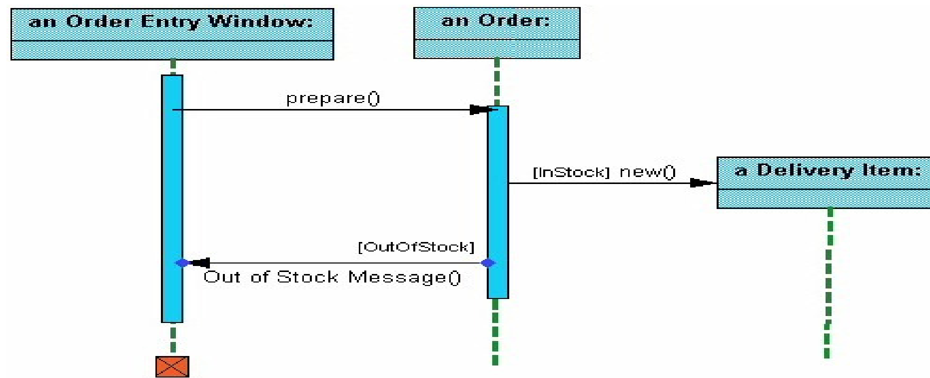


Fig 2.5 Message Sequence (c)

This simple diagram shows the sequence that messages are passed between objects to complete a use case for ordering an item.

2.6 Testing overview

2.6.1 Black Box Testing

Black box is testing without knowledge of the internal workings of the item being tested [20]. For example, when black box testing is applied to software engineering, the tester would only know the "legal" inputs and what the expected outputs should be, but not how the program actually arrives at those outputs. It is because of this that black box testing can be considered testing with respect to the specifications; no other knowledge of the program is necessary. For this reason, the tester and the programmer can be independent of one another, avoiding programmer bias toward his own work. For this testing, test groups are often used, "Test groups are sometimes called professional idiots...people who are good at designing incorrect data. Also, do to the nature of black box testing; the test planning can begin as soon as the specifications are written. The opposite of this would be, where test data are derived from direct examination of the code to be tested. For glass box testing, the test cases cannot be determined until the code has actually been written. Both of these testing techniques have advantages and

disadvantages, but when combined, they help to ensure thorough testing of the product.

2.6.2 Advantages of Black Box Testing

- More effective on larger units of code than glass box testing.
- Tester needs no knowledge of implementation, including specific programming languages.
- Tester and programmer are independent of each other.
- Tests are done from a user's point of view.
- Will help to expose any ambiguities or inconsistencies in the specifications.
- Test cases can be designed as soon as the specifications are complete.

2.6.3 Disadvantages of Black Box Testing

- Only a small number of possible inputs can actually be tested, to test every possible input stream would take nearly forever.
- Without clear and concise specifications, test cases are hard to design.
- There may be unnecessary repetition of test inputs if the tester is not informed of test cases the programmer has already tried.
- May leave many program paths untested.
- Cannot be directed toward specific segments of code which may be very complex (and therefore more error prone)
- Most testing related research has been directed toward glass box testing

2.6.4 Testing Strategies/Techniques

- Black box testing should make use of randomly generated inputs (only a test range should be specified by the tester), to eliminate any guesswork by the tester as to the methods of the function.
- Data outside of the specified input range should be tested to check the robustness of the program.
- Boundary cases should be tested (top and bottom of specified range) to make sure the highest and lowest allowable inputs produce proper output.
- The number zero should be tested when numerical data is to be input.

- Stress testing should be performed (try to overload the program with inputs to see where it reaches its maximum capacity), especially with real time systems.
- Crash testing should be performed to see what it takes to bring the system down.
- Test monitoring tools should be used whenever possible to track which tests have already been performed and the outputs of these tests to avoid repetition and to aid in the software maintenance.
- Other functional testing techniques include: transaction testing, syntax testing, domain testing, logic testing, and state testing.
- Finite state machine models can be used as a guide to design functional tests.

2.7 White Box Testing

White box testing is very different in nature from black box testing [21]. In black box testing, focus of all the activities is only on the functionality of system and not on what is happening inside the system.
Purpose of white box testing is to make sure that

- Functionality is proper
- Information on the code coverage

White box Testing is primarily development of team jobs, but now test engineers have also started helping development team in this effort by contributing in writing unit test cases, generating data for unit test cases etc.
White box testing can be performed at various level starting from unit to system testing. Only distinction between black box and white box is the system knowledge, in white box testing you execute or select your test cases based on the code/architectural knowledge of system under test.
Even if you are executing integration or system test, but using data in such a way that one particular code path is exercised, it should fall under white box testing.
There are different types of coverage that can be targeted in white box testing

- Statement coverage
- Function coverage

- Decision coverage
- Decision and Statement coverage

2.8 Review of State of Art

A **literature review** is a body of text that aims to review the critical points of current knowledge on a particular topic. Literature work regarding class diagram and sequence diagram mention below.

2.8.1 UML Class Diagram Related Work

UML class diagram related work discuss below:

2.8.1.1 UML Class Diagram Based Testing Using Verification

Lee Copeland gives this method. When performing syntax testing, this approach verifies that the class diagram contains correct and proper information [2]. There are three kinds of questions: Is it complete? Is it correct? Is it consistent?

Complete:

1. Does each class define attributes, methods, relationships, and cardinality?
2. Is each association well named?
3. Is each association and aggregation's cardinality correct?

Correct:

1. Are all attributes private?
2. Are all parameters explicit rather than being embedded in method names?
3. Do all subclasses implement the "is-a-kind-of" relationship properly?
4. Are all object states represented explicitly using states and transitions rather than as subclasses?
5. In inheritance structures, are all attributes and methods pushed as high in the inheritance structure as is proper?
6. Are all polymorphic methods within related subclasses identically named?

7. Does each association reflect a relationship that exists over the lives of the related objects?

Consistent:

1. Are each 0..* And 1..* Relationships implemented with containers/collectors?
2. Are each association's cardinalities consistent (instantaneous vs. over-time)?

Domain expert testing- after checking the syntax of the class diagrams, the second type of testing—domain expert testing comes. There are two options: either find a domain expert or attempt to become one. (The second approach is always more difficult than the first, and the first can be very hard.) In this process two kinds of questions arise: Is it correct? Is it consistent?

Correct:

1. Is each class named with a strong noun?
2. Have all redundant, irrelevant, or vague classes been removed from the diagram?
3. Is each attribute defined within the proper class? Is it of the correct type?
4. Is the visibility of each attribute correct?
5. Are the default values of each attribute specified correctly?
6. Is each attribute essential rather than computable from others?
7. Is each method in the correct class?
8. Are all method names strong verbs?
9. Does each method take the correct input parameters and return the correct output parameter?
10. Is the visibility of each method correct?
11. Does each method implement one and only one behavior?
12. Is the public interface free from unnecessary methods?

Consistent:

Is the class diagram drawn at the appropriate level: conceptual, specification, or implementation?

Trace ability testing- Finally, after having our domain expert scours the class diagrams, the third type of testing—tractability testing arises. It is to ensure to trace from the use cases and the sequence diagrams to the class diagrams and from the class diagrams back to the use cases and sequence diagrams. Again, one question arises: Is it consistent?

Consistent:

1. Is each object on the sequence diagram represented by a class on the class diagram?
2. Is every message on the sequence diagram reflected as a method in the appropriate class?

2.8.2 UML Sequence Diagram Related Work

UML sequence diagram related work discuss below:

2.8.2.1 UML Sequence Diagram Based Testing Using Slicing

This approach gives UML sequence diagram based testing using slicing. This method concentrates on dynamic slice in sequence diagram and uses that slice to generate test cases automatically from UML sequence diagram [1]. Slice consists of all statements in program Q that may affect the value of V at some point.

Dynamic slice contains of any those part of a sequence diagram that actually affect the value of a variable at a message point for a given execution or condition. Generate test data with respect to slice and main aim is adequate test coverage without unduly increasing the number of test cases.

It has another that is dependency graph that is drawn on the basis of node dependency. Node means message points in interaction of objects. This graph draws using two terms i.e. USEVAR (X) and DEFVAR (X). USEVAR (X) means it is set of all nodes that use the value of variables. DEFVAR (X) means it is set of all nodes that define the variables. So with help of this method we

generate automated test cases from sequence diagram and it is a cluster level testing where object interaction are tested.

2.8.2.2 Sequence Diagram Testing Using Verification

Lee Copeland gives this method while performing syntax testing; this approach verifies that the sequence diagram contains correct and proper information [3]. Three kinds of questions arise: Is it complete? Is it correct? Is it consistent?

Complete:

1. Does each object required for the interaction appear on the diagram?

Correct:

1. Have all objects not required in the interaction been removed from the diagram?
2. Does each object's lifeline begin and end at the proper time?
3. Is each object's activation described properly?
4. If the object's lifetime terminates, is it indicated with an X?
5. Is each message well named with a strong verb?
6. Are proper parameters included for each message?
7. Are conditional branches drawn properly?

Consistent:

1. Do conditionals cover all of the cases?
2. Have any overlaps of conditionals been removed?

Domain expert testing, after checking the syntax of the sequence diagrams, we proceed to the second type of testing—domain expert testing. Again, we have two options: find a domain expert or attempt to become one. (The second approach is always more difficult than the first, and the first can be very hard.) Continuing, we ask two kinds of questions: Is it complete? Is it correct?

Complete:

1. Are all the ways that things could go right identified and handled properly?
2. Are all the ways that things could go wrong identified and handled properly?
3. Does the main success scenario run from the trigger to the delivery of the success end condition?

Correct:

1. Does the sequence diagram show each step that must be executed to implement the function?
2. Can each step actually be implemented?

Trace ability testing finally, after having our domain expert scour the sequence diagrams, the third type of testing—trace ability testing comes. It is to ensure to trace from the use cases to the sequence diagrams and from the sequence diagrams back to the use cases. Again, one question arises: Is it consistent?

Consistent:

1. Is each use case represented by at least one sequence diagram?
2. Does each actor appear on at least one sequence diagram?

2.8.2.3 UML Sequence Diagrams And State Diagrams

Dehla Sokenou gives this method. It generates test cases with combination of UML sequence diagram and state chart diagram [5]. The main information is extracted from sequence diagrams, which is complemented by initialization sequences for the participating objects derived from state diagrams.

One of the main problems in testing is how to select test cases in object-oriented programming. It is impossible to stimulate the program with all data of the input domain. A pragmatic approach is to concentrate on typical message sequences as modeled using the sequence diagram. Testing based on sequence diagrams seems to be intuitive. Each sequence diagram specifies one test case or set of test cases. But normally, modeled sequences are incomplete, and offer no information about

the time in the program's life cycle when the modeled behavior will occur nor state information about participating objects. A test based on sequence diagrams must consider these issues. This approach uses state diagrams to obtain the required information. Each sequence diagram is considered as a set of test cases. An attached state diagram for each participating object defines its states. Each combination of initial state configurations defines at least one test case in the set. UML models can be interpreted differently depending on their application. Thus, the UML defines a set of semantic variation points. For testing purposes, the variation points and concentrate on sequence diagrams and state diagrams are determined. This approach assumes that models are consistent. If not, the test case generation will result in an error in most cases. It is not necessary to provide all models as input for test case generation. This approach uses UML protocol state machines for specifying object life cycles. Protocol state machines differ in some points from behavioral state machines, the other kind of state diagram in UML. Protocol state machines do not define actions, but it is possible to specify post conditions of transitions instead. This approach assumes that events in the state machine are method calls (call events). Methods referenced in the protocol state machine are update methods, all other methods being query methods. The set of states in which an update method triggers a transition defines an implicit precondition for these methods, meaning that the call of an update method in all other states violates the precondition. In sequence diagrams, only messages in the form of method calls, focusing on synchronous communication between objects are considered.

Chapter 3
Problem Statement

Problem statement describes the gap in the existing work and problem formulation. The Gap in existing work shows, what are the limitation in the existing work and which technique they are used. In problem formulation, we give appropriate solution to solve the existing problem and suggest the novel work.

3.1 Gaps in Existing Work

In **"UML Class Diagram Based Testing Using Verification"** testing performed with the help of design review and it tells about how to verify design. It has syntax testing, domain expert testing and trace ability testing. It does not deal with automated testing. It does not have any test case generation technique.

"**UML Sequence Diagram Based Testing Using Slicing",** this method concentrates on dynamic slice in sequence diagram and uses that slice to generate test cases automatically from UML sequence diagram. But these dynamic test cases depend upon variable value and its change affect with respect to the message sequence.

"UML Sequence Diagram Testing Using Verification" verifies that the sequence diagram contains correct and proper information. It contains syntax testing, domain expert testing and trace ability testing. It does not specify test case generation. It is not an automated testing.

"UML Sequence Diagrams and State Diagrams" generates test cases with combination of UML sequence diagram and state chart diagram. This approach uses state diagrams to obtain the required information. Each sequence diagram is considered as a set of test cases. It is assumed that models are consistent. If not, the test case generation will result in error in most cases.

3.2 Problem Formulation

Generally testing is performed on coding part, which contain statements, loops, classes and functions. This type of testing cannot fix problems because if we change the executable statements of software then number of test cases also change, because we check logical part of software and plan the test case. Like if a program has two loops then its test cases are to test the functionality of only these two loops, if we add another loop then total loops are three and its test cases will also increase. That means increase in statements leads to rise in complexity, which will affect the test plan and test case structures.

Same in object oriented programming methodology, it contains classes and functions. We want to change class scope that means public, private, protected or we want to add new class that is inherited from or dependant upon other classes then it is very difficult task to find which module is affected by this change and what is the affect on other modules. It is very difficult to identify test cases required to fix this problem and also to know whether these test cases are optimal or not. One solution is we use code review to fix that problem but it's not efficient because each programmer has own style of coding, so coding review is difficult task. Another alternative is to perform dynamic testing that means run software and check behavior of software. But these procedures again indirectly focus on coding review for finding errors or change effect.

Coding review is not much efficient method, which is used in structured and object oriented approach. So we proposed a novel solution that is design based testing. In software development life cycle design part first than coding or implementation part. Testing problem should be applied on design rather than coding because a software design is converted into code. So if problem occur in design, code will be error full.

To perform design based testing, we can use Unified Modeling Language (UML). UML has different design diagrams that are used to specify the software behavior i.e. static and dynamic. UML diagram methodology is applied on object-oriented approach, not on structured approach.

Many researchers and practitioners are working on UML based testing. UML based testing is independent of code; it specifies general behavior of software. This

type of design easily deployed on any languages like c++, java etc. We can perform all testing techniques on UML design. UML deals with different diagrams like class, sequence, activity, state chart, component diagram, object, collaboration diagram. All these diagrams used for testing.

With help of design we can produce test cases from UML diagrams. All diagram related with each other, they can be combined them and testing can on of diagrams. But they are some limitations like:

- UML is a graphical view of software. If an automated testing is performed on UML design, it is difficult. Automated testing means automatically generate test cases from deign.
- How to combine two UML diagram and perform testing i.e. how we can pass the information from one UML diagram to another UML diagram for efficient.
- Finding static test cases and dynamic test cases from UML diagrams. UML diagram has two-type behavior i.e. static and dynamic. So finding static test cases and dynamic test cases that are optimal solution.
- Perform regression testing on the design and finding affect of changes on the other modules with automated testing.

So these limitations can be resolved with help of proposed system. In the proposed system, testing is performed on two diagrams i.e. class diagram and sequence diagram. Proposed system focus on these two diagrams and with the help of these diagrams we can find out optimal solution and perform better testing than previous approach.

Class diagram based testing solved the problem of static test cases with automated method and sequence diagram based testing solved the problem of dynamic test cases. With help of UML, we can perform regression testing i.e. finding affect of change in design.

Proposed work divided into modules that perform automated testing on design. The Modules are class analyzer, function analyzer, static test cases generator, sequence diagram based testing, dynamic test cases generator and regression testing module.

Chapter 4
Proposed System And Implementation

4.1 Introduction

The proposed system focuses on the two UML diagrams.

- Class diagram
- Sequence diagram

Various types of information is extracted from these diagrams to generate test cases. For static information class diagrams are used and for dynamic type of information sequence diagrams are used.

4.1.1 Class Diagram

With the help of class diagram we find out: -

- Classes
- Relationship between classes
- Dependency
- Parent/child relationship
- Functions and its scope
- Global and local variables

4.1.2 Sequence Diagram

With the help of sequence diagram we can find out: -

- Object interaction
- Class interaction
- Dynamic behavior of software

4.1.3 Combination of Class and Sequence Diagram

Combination of Class and Sequence diagram means, we can use class-based information and sequence diagram to generate test cases. These test cases specify dynamic behavior of software and show the interaction between two classes using functions.

4.2 Brief Overview About System

In UML class diagram testing, it finds out dependency constructs, with the help of these constructs, we can draw class based flow graph and generate static test cases automatically.

UML sequence diagram aid to generate dynamic test cases with collaboration of class-based constructs and sequence diagram constructs. These test cases are optimal test cases to test design.

In UML based regression testing, the effect of change on the design is studied. Class diagram based constructs are used to find the effect of change in design.

4.3 UML Class Diagram Based Testing

4.3.1 Class Analyzer

Class Analyzer deals with design based source code. This code helps to produce some important constructs like classes, its dependency, parent and child relationships between classes, functions regarding each class and its scope.

Another important point is Class Analyzer gives a Graph, called as Class Graph. This graph shows dependency between classes shown in Fig 4.1.

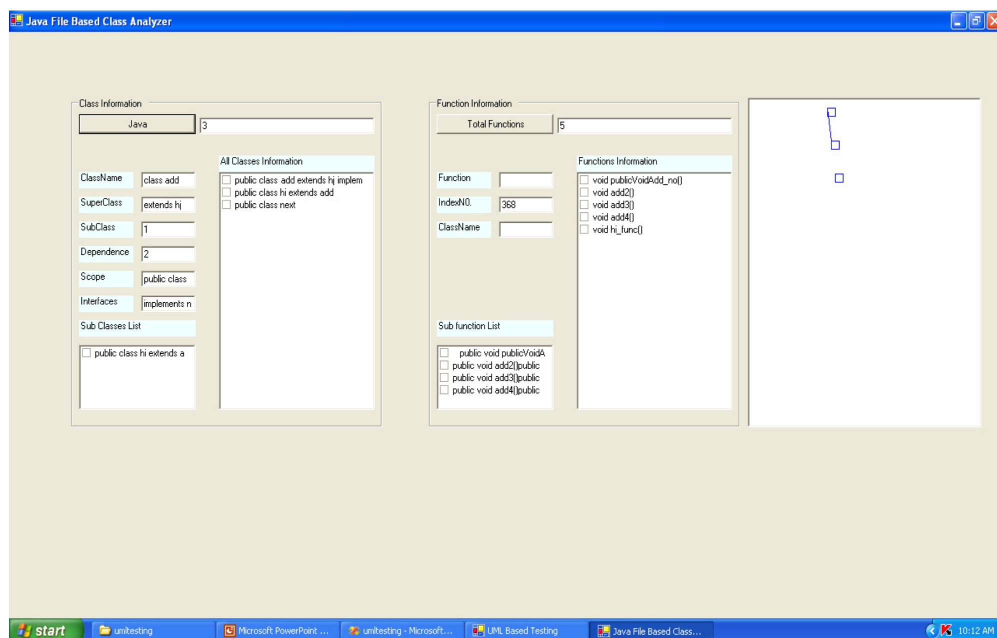


Fig 4.1Class Analyzer

Class Analyzer works on the design based file that has constructs of classes, function and attributes. This file is generated by Rational Rose and contains design based constructs. So this file is passed to Class Analyzer tool that finds out design related constructs. As soon as that tool extracts classes, a corresponding graph is generated. The graph shows dependency between classes.

This tool also helps to find out variables in a class i.e. local and global and also finds out scope, interface, super class, subclass, dependency and its related functions.

Class Analyzer provides this information and draws graph, which is made up of class dependency (parent/child relationships) and interfaces used in this design. So this information is used to verify the design construct because class based code or class diagram based code deals with static behavior of design.

Class Graph gives a general visualization of interaction between classes likes if we take the java code, Example-4.1:

```
Class abs
{
Int a, b, c;
Public void pass ()
{
   }
}//end abs
Class cal extends abs
{
Public void use ()
{
  }
}//end cal
```

Example-4.1

In this code we have two classes class ABS and class CAL. Class CAL use functionality of ABS class that means CAL class depends upon ABS class or we can say that CAL class inherits some functionality of ABS class. Dependency flow graph is shown in fig 4.2.

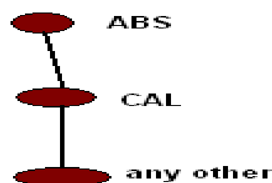


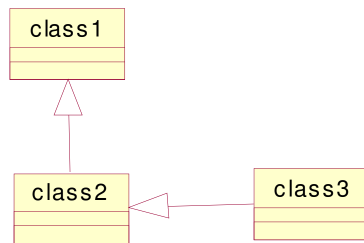
Fig 4.2 Dependency Flow Graph

4.3.2 Details of Class Analyzer

We have three classes in a program that is class1, class2 and class 3. Its design-based code is given below:

```
Public Class 1
{   };
Public class 2 extends class1
{  };
class 3 extends class2
{  };
```

This design-based code depends upon UML class diagram, the UML class diagram is as shown in Fig 4.3 and class path graph is given in Fig 4.4. This class path is made on the basis of dependency with classes.



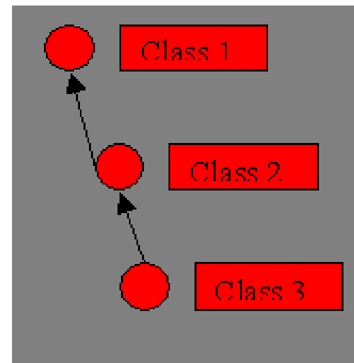


Fig 4.3 UML Class Diagram   Fig 4.4 Class Path Graph

4.3.3 Class Path Mechanism

- To test class2, its dependency needs to be checked.
- It is dependent on class1. We draw link between them that means class2 functionality depends upon class1.
- It means if we test class2 then we need to test class1 also.
- So its test case looks like CLASS1->CLASS2 OR CLASS2->CLASS1 (static test cases).

4.4 Function Analyzer

Function Analyzer uses is design based source code. Functional analyzer generates more important constructs than Class Analyzer. Functional analyzer has further detailed information regarding classes and functions for example classes, its functions, global variables in a class, which variable used by which function, scope of variables, classes and functions based Graph. Function based graph depicts the class relationships including class to class interaction and class to function interaction.

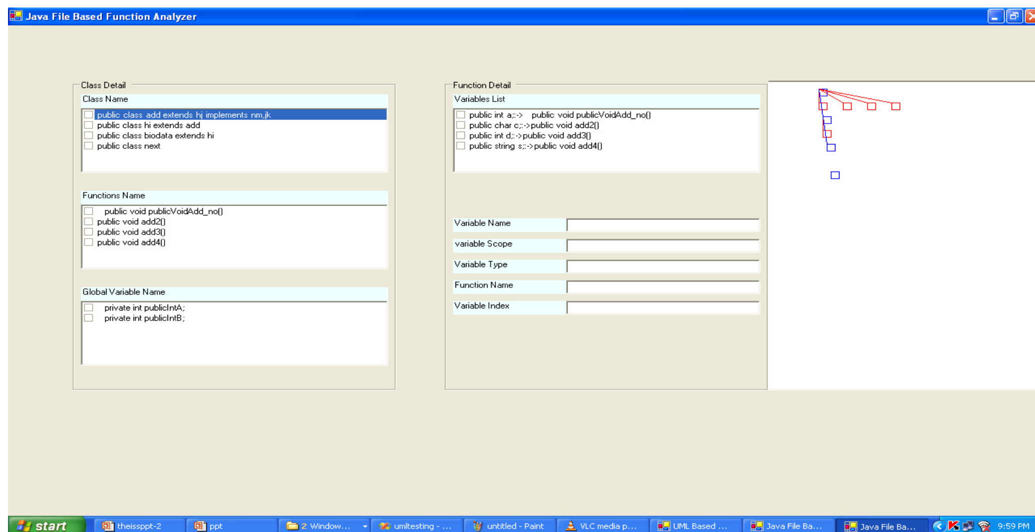


Fig 4.5 Function Analyzer

With the help of classes and functions based graph, we concentrate on classes, its functions and global variable regarding particular class. It also depicts which variables are used by which functions. All this information is important for testing any particular class, its functions that is unit testing and this information is important for regression testing because it contains which variable is used by which function. With the help of this information we can find out the effect of changes on the other module like if someone changes scope of variables, what will be the effect of that variable on the other classes and functions in case of dependency. Class and function based graph is shown in Fig 4.5 and Fig 4.6.

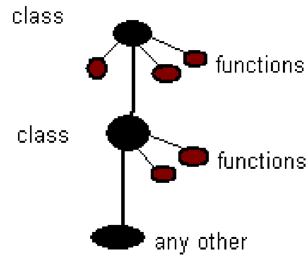


Fig 4.6 Class/Function Flow Graph

Function Analyzer tool further refines Class Analyzer that means it totally concentrates on function detail and draws graph of class and its functions. This tool uses design based code file that contains constructs of design.

4.4.1 Class Based Test Cases

Class based test cases generally generates static test cases. Test cases information is extracted from Class Analyzer and Function Analyzer. These test cases are based upon the function present in that class

.

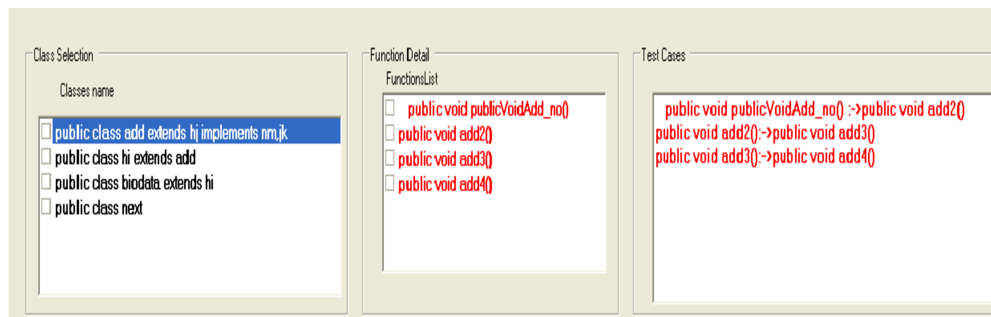


Fig 4.7 Static Test Cases

Generally test cases provide a track to test particular class or group of classes. In case of design we specify the classes and functions so test case structure is like that CLASS NAME -> FUNCTION NAME -> other module or FIRST CLASS NAME -> SECOND CLASS NAME -> SO ON. Or FIRST FUNCTION NAME -> SECOND FUNCTION NAME -> SO ON. Consider example of above java code that has two classes and CLASS ABS -> VOID PASS FUNCTION -> CLASS CAL.

But problem is that these test cases are static test cases, means its path or sequence of test cases depends upon permutation or combination mechanism. We cannot

predict optimal test cases from the static behavior extracted from class diagram. Due to this, number of test cases increase as combinations increase so we can neither perform optimal testing on software design nor optimal test cases can be generated to test a particular software design. This problem is solved with the help of dynamic behavior of software design. So sequence diagram is proposed to test the dynamic behavior.

4.4.2 Details of Function Analyzer

Function Analyzer deals with classes and its functions. We have a design-based code that has three classes and its functions. Function Analyzer first finds classes and then functions of its classes. Using following design based code:

```
Public Class 1
{Void   date ()
{}};
Public class 2 extends class1
{Void use_date ()
{}
Void show_date ()
{}
Void modify_date ()
{}};
Class 3 extends class2
Void use_modifydate ()
{}
Void show_modifydate ()
{}
{};
```

Example-Class1 4.2

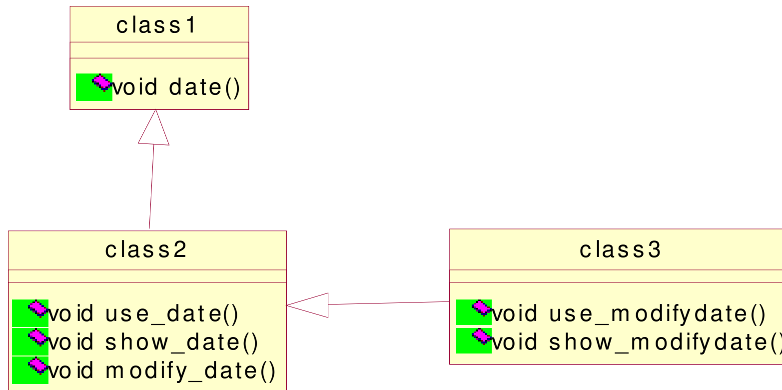


Fig 4.8 UML Class Diagram

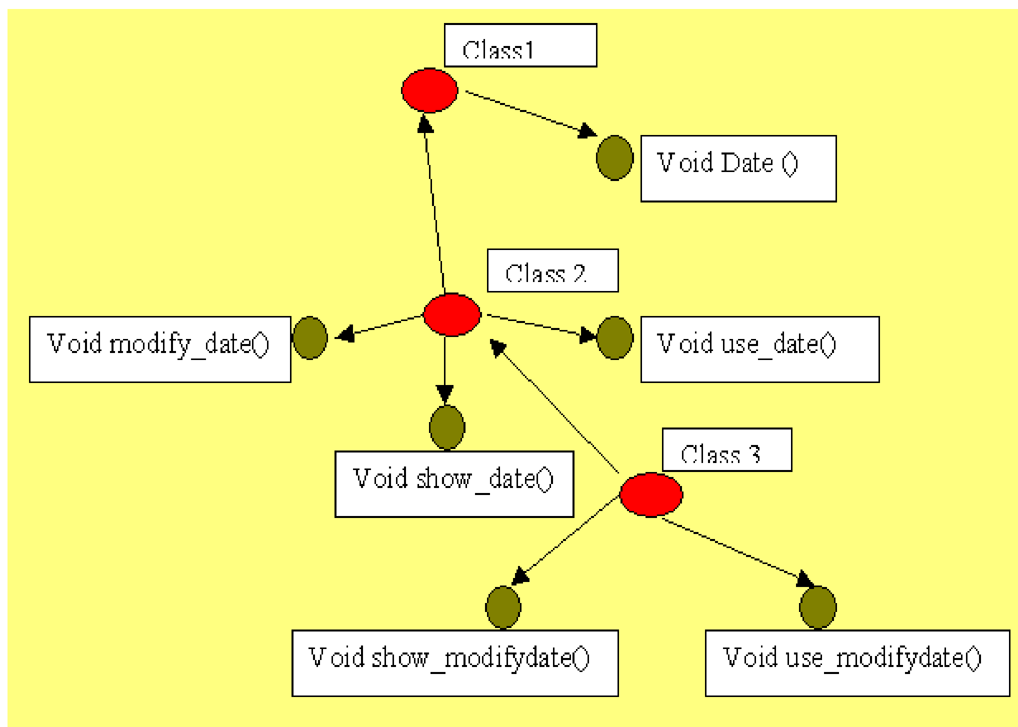


Fig 4.9 Class and Function Path

4.4.3 Function Analyzer Mechanism

- To test class2 we need to find out its dependency that is class1.
- After this we need to find out functions of each class.
- Test class2 then we following test case is generated:

  CLASS2->VOID DATE ()->CLASS1->VOID USE_DATE ()->VOID SHOW_DATE ()->VOID MODIFY_DATE ().

  This test case is combination of two classes. .

- To test individual class2 test case is CLASS2->VOID USE_DATE ()->VOID SHOW_DATE ()->VOID MODIFY_DATE ().

4.5 Sequence Diagram And Dynamic Test Cases

Sequence diagram depicts dynamic behavior of software between classes or objects. But problem here is how to produce dynamic test cases from sequence diagram automatically. The test cases use information produced by Class Analyzer and Function Analyzer.

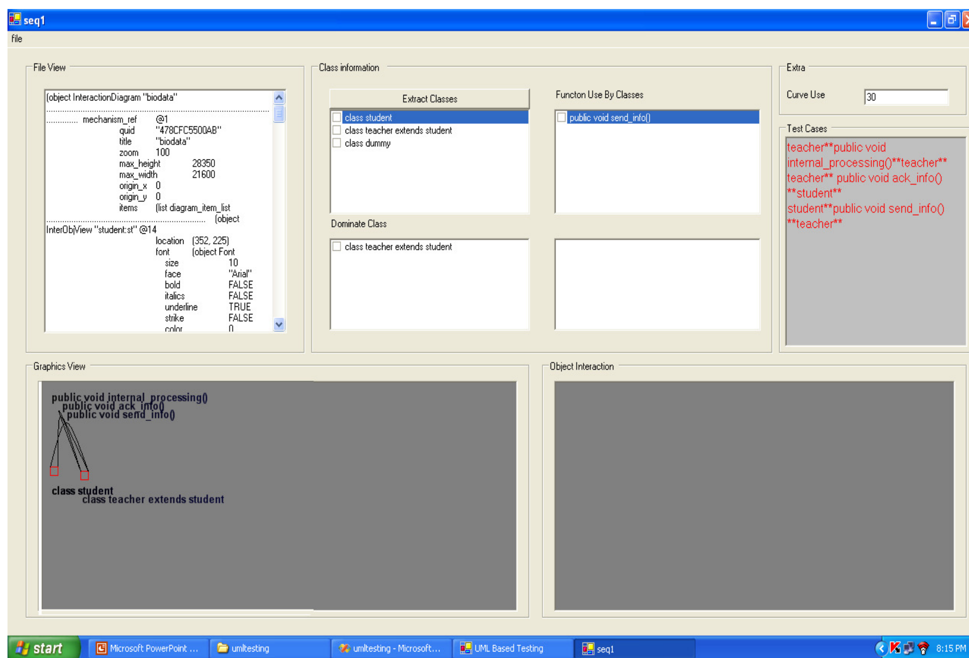


Fig 4.10 Sequence Testing Module

To solve this problem, we use UML Sequence Diagram based on MDL format file of sequence diagram. From MDL file, we produce some important constructs and combine them with Class Analyzer and Function Analyzer and then we generate dynamic behavior of sequence diagram with a Graph, shown in figure 4.10. With help of this, we generate dynamic test cases to test the dynamic behavior of software.

4.5.1 Generation of Dynamic Test Cases

In dynamic test cases are produced on the basis of Sequence Diagram, MDL format and Class & Function Analyzer. Dynamic Test Cases show the function calling sequence i.e. which function is called by which class.

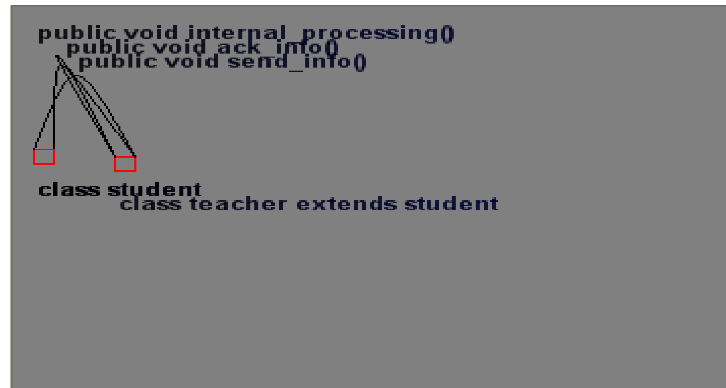


Fig 4.11(a) Dynamic Test Cases

teacher**public void
internal_processing()**teacher**
teacher** public void ack_info()
student
student**public void send_info()
teacher

Fig 4.11(b) Dynamic Test Cases

Dynamic test cases show actual runtime behavior in a design or software. Like in this diagram, we have two classes i.e. class student and class teacher, shown in figure 4.1. Class teacher uses functionality of student. Class student has one public void send_info () function. This function is calling teacher class. It means student class generate a call to teacher class with help of void send_info ().

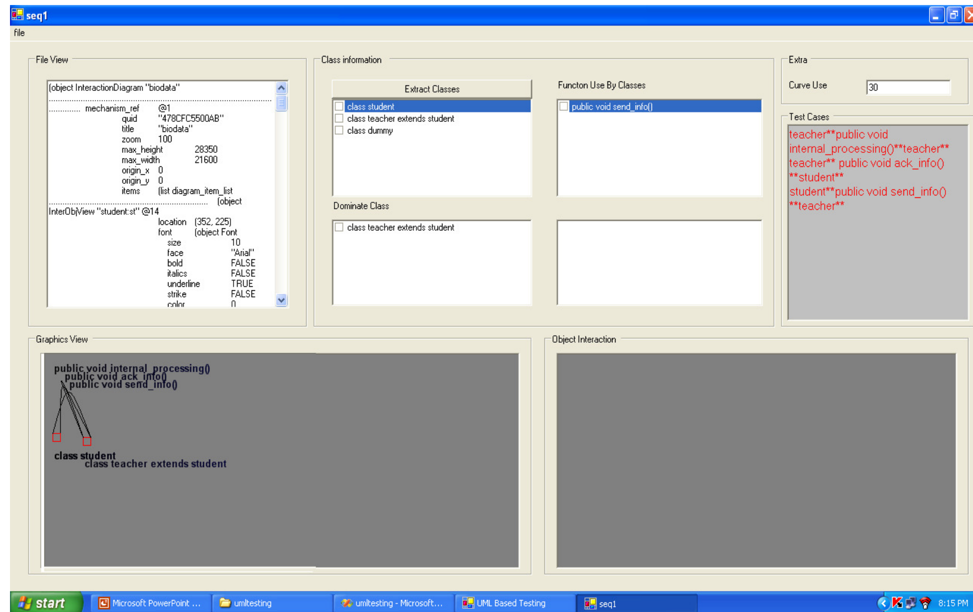


Fig 4.12 Dynamic test cases & sequence diagram

Teacher class has two functions, void ack_info () and void internal processing (). Function void internal processing () is used for internal processing by any object of class teacher object. Void ack_info () function used by both teacher class and student class like as teacher class object generate or activate this function and inform student class or call student class, so it is like producer and consumer problem, all communication are shown in Fig 4.12.

If we carefully observe the following dynamic test cases that are automatically generated.

- TEACHER ** PUBLIC VOID INTERNAL_PROCESSING ** TEACHER.
- TEACHER ** PUBLIC VIOD ACK_INFO ** STUDENT.
- STUDENT ** PUBLIC VOID SEND_INFO () ** TEACHER.

They are three test cases, which visualize the behavior of dynamic processing between classes.

So these test cases are optimal because these test cases are not based on permutation and combination. It totally shows the actual processing between objects and classes. So it is better than class based testing because it's depend upon static test cases and combination method. It gives us better solution than class based testing.

4.6 UML Design Based Regression Testing

Regression testing means testing effect of change in one module to another module. To perform regression testing is mostly performed on coding part. But classes, functions and variables are defined beforehand. A design-based regression testing approach is also proposed. This proposed technique helps to identify any change, if occurs during testing and debugging or during updation.

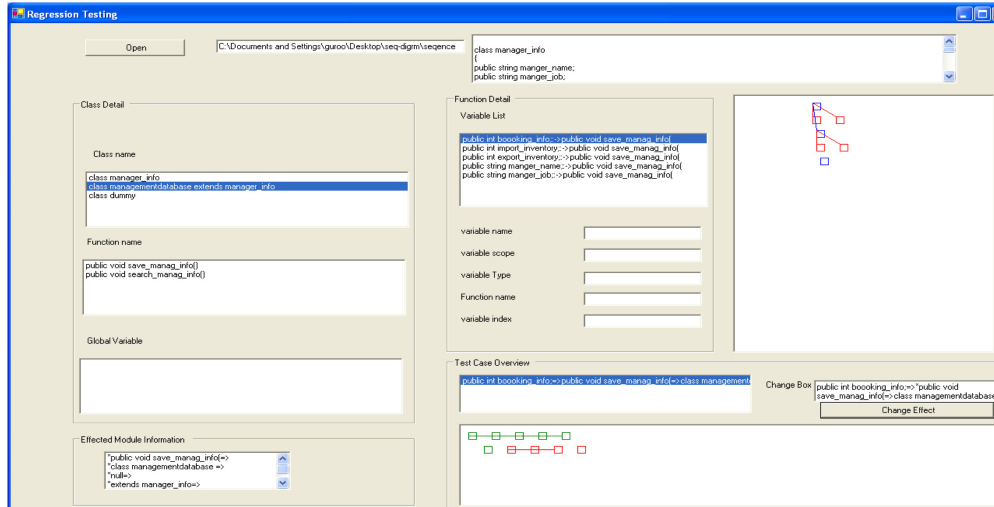


Fig. 4.13 Regression Testing

Shown in Fig 4.13 Design based regression testing module. That module automatically finds out effect of change, that means where change occurred and how many modules are affected by change. We have an example shown below:

```
Class biodata
{
Public string name;
Public int age;
Public int salary;
Public void Add_biodata ()
{}
Public void Show_biodata ()
{}};
Class database extends biodata
{
Public void submit ()
{}
Public void display ()
```

```
{}
Public void search ()
{}};
```

Example – Class Biodata 4.3

Here we have two classes i.e. biodata and database. Database uses functionality of Biodata class. Class biodata has two functions i.e. public void add_biodata (), public void show_biodata and three variables. Class database has three functions public void submit (), void display () and void search (). If we want to generate test cases from this UML design based code, to test Database class then test case is BIODATA -> PUBLIC VIOD ADD_BIODATA -> DATABASE -> PUBLIC VIOD SUBMIT -> ANY OTHER. If we want to test particular variable then test is PUBLIC STRING NAME -> BIODATA -> PUBLIC ADD_BIODATA -> DATABASE -> ANY OTHER.

If we focus on variable change effect i.e. public string name of Biodata class that means variable used by Database class, so its test case is PUBLIC STRING NAME -> BIODATA -> PUBLIC ADD_BIODATA -> DATABASE -> ANY OTHER.

This is test case is totally correct and run able but if we change the scope of variable like public to private, so test case will be PRIVATE STRING NAME -> BIODATA -> PUBLIC ADD_BIODATA -> DATABASE -> ANY OTHER. Now test case has changed. It affects inherited class that is Database class because Database class cannot access the private data members, so its changed form is PRIVATE STRING NAME -> BIODATA -> PUBLIC ADD_BIODATA -> ANY OTHER. And its effected modules are class database and its function. This information is very important because it tells, if we change the scope of variable then what are effects on the other modules. Same thing we apply on global variables and local variables or any other function and related class.

The main advantage is identification of change impact. So there is no need to start regression testing from scratch, only affected modules need to be tested again. This change impact information will be future help to plan test cases.

4.7 Working of UML Based Testing Tool

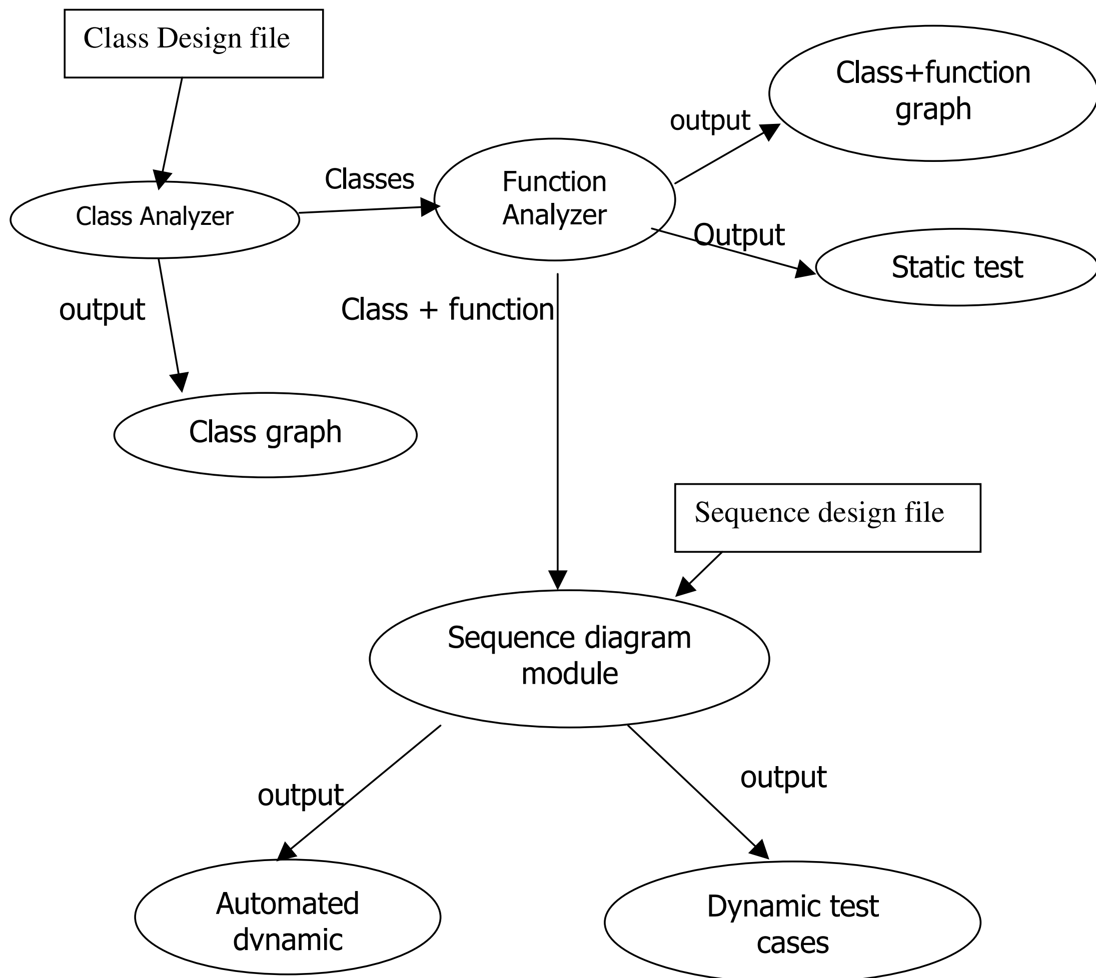


Fig 4.14 Overall Working of System

This flow diagram shows the overall working behavior of Class Analyzer and Function Analyzer and also shows which type of input we need and which type of output, Class Analyzer and Function Analyzer generates.

4.7.1 Working steps

- Input class design file to Class Analyzer. The Class Analyzer finds classes and class- flow graph based on dependency.

- Classes are passed to Function Analyzer. Function Analyzer module finds functions and variable of each class and generates a class & function graph. Its important output is static test cases.
- Classes & functions based file and sequence design file are passed to sequence testing module. It generates a sequence diagram like flow graph and dynamic test cases.

4.8 Working of Regression Testing Tool

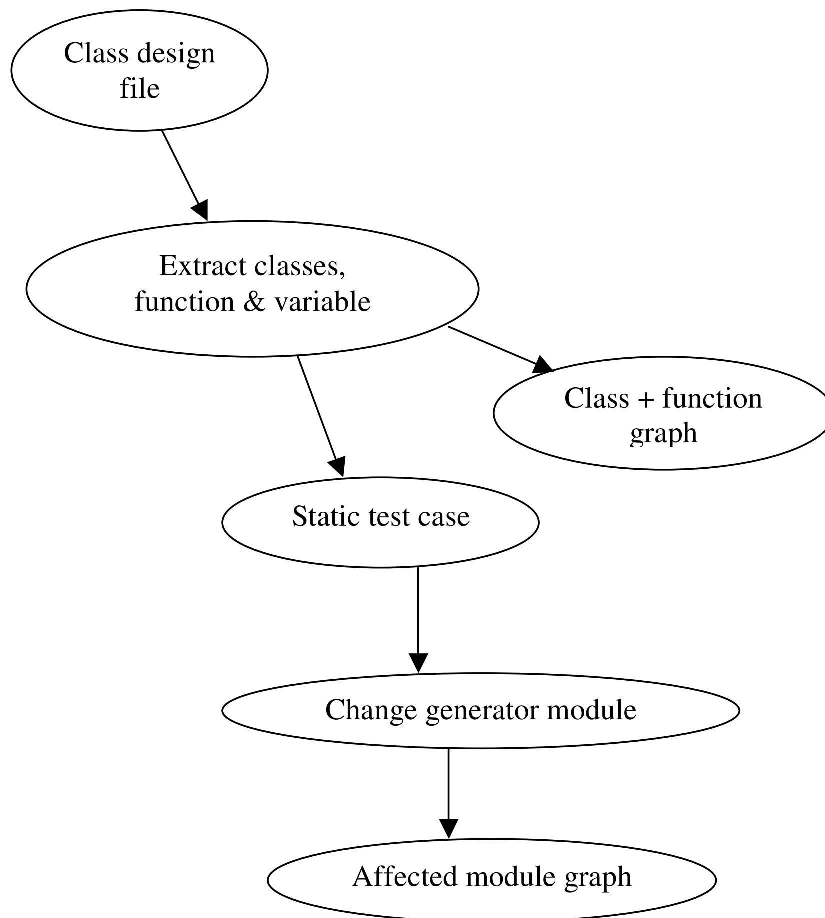


Fig 4.15 Regression Testing Overview

This flow diagram shows the working behavior of regression testing module. It specifies input and output of regression testing.

4.8.1 Working Steps

- Input class diagram based file, to a file reader.
- File reader finds out constructs like classes, function, and variables.
- Generate a class and function flow graph based on dependency.
- Having performed above activities, program generates static test cases.
- If we change any test cases then change generator module gives output in the form of affected module

Chapter 5
Conclusions & Future Work

To perform design based testing, we need a language that can deal with the design efficiently i.e. Unified Modeling Language (UML). UML consists of different designs that are used to specify the static and dynamic behavior of the software. UML diagram methodology is applied on object-oriented approach, not on structured approach. In the proposed system testing is performed on two diagrams i.e. class diagram and sequence diagram. Proposed system focuses on these two diagrams and with the help of these diagrams we can find out optimal solution and perform better testing than previous approach.

Class diagram based testing solves the problem of static test cases with automated method and sequence diagram based testing solves the problem of dynamic test cases. Further with help of UML we can perform regression testing i.e. finding affect of change in design.

The proposed work is divided into several modules that perform automated testing on design. The modules are class analyzer, function analyzer, static test cases generator, sequence diagram based testing, dynamic test cases generator and regression testing module.

5.1 Conclusions

- Design based code of class diagram is used to generate efficient test cases automatically for static behavior
- Collaboration of mdl code of sequence diagram and design based code of class diagram has been used to generate test cases automatically for dynamic behavior
- This has been observed that proposed tool successfully gives change impact for regression testing

5.2 Future Work

- Design based testing constructs produced from class and sequence diagram testing module can be further extended to use state chart diagram and activity diagram to generate test cases.

- Further work can be explored to use formal methods to make system suitable for real and large systems.
- This technique further can be extended to include all types of software artifacts to predict change for regression testing.

References

[1] Philip Samuel, Rajib Mall and Sandeep Sahoo, UML Sequence Diagram Based Testing Using Slicing, IEEE Indicon 2005 Conference, Chennai, India, I I 'I 3 Dec. 2005, pages 176-178.

[2] Lee Copeland class diagram based testing, http://www.stickyminds.com.

[3] Lee Copeland sequence diagram based testing, http://www.sticyminds.com.

[4] Rumbaugh, Jacobson, and Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, Reading, MA, 1999.

[5] Dehla Sokenou, Generating Test Sequences from UML Sequence Diagrams and State Diagrams, GEBIT Solutions GmbH.

[6] Beizer, Boris, Software Testing Techniques, Second Edition. Van Nostrand Reinhold, 1990.

[7] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: Object-Oriented Modeling and Design. Prentice Hall, 1991.

[8] D. Seifert, S. Helke, and T. Santen. Test Case Generation for UML Statecharts. In PSI'2003.

[9] A. Bertolino and F. Basanieri, "A practical approach to UML-based derivation of integration tests", Proceedings of the 4th Intemational Software Quality Week Europe and Intemational Intemet Quality Week Europe, Brussels, Belgium, 2000, QWE.

[10] B. Korel and J. Rilling, "Dynamic program slicing methods", Infonnation and Software Technology, 40:647 659, 1998.

[11] J. Offutt and A. Abdurazik, "Generating tests from UML specifications", Proceedings ofthe 2nd Intemational Conference on UML, Lecture Notes in Computer Science, volume 1723, pages 416 – 429, Fort Collins, TX, 1999. Springer-Verlag GmbH.

[12] B.Jeng and E. J. Weyuker, "A simplified domain-testing strategy", ACM Transactions on Software Engineering and Methodology (TOSEM), 3(3), 1994.

[13] B. Korel, "Automated software test data generation", IEEE Transactions on Software Engineering, 16(8): 870 - 879, 1990.

[14] Clay E. Williams "Software Testing and the UML" Center for Software Engineering, IBM T. J. Watson Research Center.

[15] Poston, R. Automated test from object models. CACM 37,9 (Sept. 1994), 48-58.

[16] Perry, D. and Kaiser, G. adequate testing and object-oriented programming. JOOP (Jan./Feb. 1990), 13-19.

[17] UML & Rational, http://www.ibm.com/developersworks/rational

[18] UML Modeling http://agilemodeling.com

[19]Testinghttp://atlas.kennesaw.edu/dbraun/csis4650/A&D/UML_tutorial/resources.htm

[20] Black Box Testing http://www.cse.fau.edu/~maria/courses/cen/Black1.html

[21] White Box Testing http://testtinggeek.com

List of Papers/Publications

[1] Gurpreet Singh and Rajesh Kumar Bhatia "UML Design Based Testing" "ACM/IEEE 11th International Conference on Model-Driven Engg. Langauges and Systems (Formerly UML)", Toulouse, France, 28 Sep-2 Oct (Communicated).

[2] Gurpreet Singh and Rajesh Kumar Bhatia "UML Design Based Testing" "International Symposium on Computer Science and Its Application", Australia, 14 -16 Oct (Communicated).