DESIGN OF TEST CASE PRIORITIZATION TECHNIQUES FOR OBJECT ORIENTED SYSTEMS

THESIS

Submitted in fulfillment of the requirement of the degree of

DOCTOR OF PHILOSHOPHY

to

J. C. BOSE UNIVERSITY OF SCIENCE & TECHNOLOGY,

YMCA, FARIDABAD

by

VEDPAL

Registration No: Ph.D-12-2K12

Under the Supervision of

Dr. NARESH CHAUHAN

PROFESSOR

Department of Computer Engineering

Faculty of Engineering and Technology

J. C. Bose University of Science &Technology, YMCA, Faridabad

Sector-6, Mathura Road, Faridabad, Haryana, INDIA

APRIL, 2019

Dedicated

to

My Parents

CANDIDATE’S DECLARATION

I hereby declare that this thesis entitled “**DESIGN OF TEST CASE PRIORITIZATION TECHNIQUES FOR OBJECT ORIENTED SYSTEMS”** being submitted in fulfillment of requirement for the award of Degree of Doctor of Philosophy in the Department of Computer Engineering under Faculty of Engineering and Technology of J. C. Bose University of Science and Technology, YMCA, Faridabad, during the academic year March 2013 to April 2019, is a bonafide record of my original work carried out under the guidance and supervision of **DR. NARESH CHAUHAN, PROFESSOR, DEPARTMENT OF COMPUTER ENGINEERING** and has not been presented elsewhere.

I further declare that the thesis does not contain any part of any work which has been submitted for the award of any degree either in this university or in any other university.

(VEDPAL)

Registration No: Ph.D-12-2K12

CERTIFICATE

This is to certify that the thesis titled **"DESIGN OF TEST CASE PRIORITIZATION TECHNIQUES FOR OBJECT ORIENTED SYSTEMS"** submitted in fulfillment of the requirements for the award of Degree of Doctor of Philosophy in Department of Computer Engineering under Faculty of Engineering and Technology of J.C. Bose University of Science & Technology, YMCA, Faridabad, during the academic year March 2013 to April 2019, is a bonafide record of work carried out under my guidance and supervision.

I further declare that to the best of my knowledge, the thesis does not contain part of any work which has been submitted for the award of any degree either in this university or in any other university.

DR. NARESH CHAUHAN
Professor
Department of Computer Engineering
Faculty of Engineering and Technology
YMCA University of Science & Technology Faridabad

Date:

The Ph.D. viva-voce examination of Research Scholar Vedpal (Ph.D – 12-2K12) has been held on ………………………….

(Signature of Supervisor) (Signature of Chairman) (Signature of External Examiner)

ACKNOWLEDGEMENTS

First of all, I would like to thank **God**, the Almighty, for providing enough courage and his blessings to me for completion of this thesis. I have experienced your guidance every day.

I would like to express my sincere gratitude to my thesis supervisor, **Dr. Naresh Chauhan**, for his tremendous guidance. I would like to thank him for encouraging my research work and for allowing me to grow as a researcher. The advices given by him related to my research work as well as on my career have been priceless.

I am grateful to Late **Prof. A. K. Sharma** for his encouragement and motivation for the completion of this work. I would also like to special thank **Dr. C.K. Nagpal**, **Dr. Komal Kumar Bhatia**, **Dr. Atul Mishra** and **Dr. Manjeet Singh** for their valuable advices and motivation for accomplishing this task.

I am also very thankful to **Dr. Harish Kumar**, **Dr. Neelam Duhan** and **Dr. Rashmi Popli** for their continuous support and encouragement to complete my research work. I would like to thank **Sh. Sushil Kumar , Sh. Umesh Kumar, Sh. Krishan Bhardwaj , Sh. Neeraj Parashar, Sh. Sanjay** for always providing their support and necessary resources for the completion of this thesis.

I also want to deliver my thanks to faculty members of Computer Engineering and Computer Applications departments for their support and cooperation. Although it is not possible to name individual, I cannot forget my well-wishers for their persistent support and cooperation. I am also thankful to all my students who helped me directly or indirectly in completing my research work.

A special thanks to my family. It is not possible to explain in words that how I am thankful to my mother Late **Smt. Sheela Devi**, my father, Sh**. Raghubir Singh**, my uncle Late **Sh. Om Prakash**, aunty **Smt. Raja Bala**, mother – in law **Smt. Raj Kumari** and my father – in- law **Sh. Pawan Kumar.** I am grateful to my brother in law Sh. Raj Kumar for his blessings.

I would also like to thank my wife **Sonika** for her support. Finally I want to express my thanks to my beloved daughter **Mishita** for being a good girl always cheering me up.

Thanks to all of you!

(VEDPAL)

ABSTRACT

Software Testing is an important activity during the development of the software. It helps to ensure that the developed software provides all the functionality in an efficient manner as desired by the customer. In past three decades, the object oriented programming system is preferable for developing the software due to its features. However all the concepts of conventional testing loses their meaning in the testing of the object oriented software. The testing of the object oriented software has various issues like basic unit for testing, inheritance, polymorphism, white box testing, integrated strategies, etc.

Further the testing of the object oriented software consumes a lot of time, efforts and resources. To ensure the quality of software, the efficient test cases are designed and executed. It is very difficult and costly to execute the large number of test cases. The test cases should be executed in such a way that they find the maximum faults at earlier stages. It is very costly and time consuming to detect and fix the bugs at later stages. So the test cases should be ordered to detect the maximum faults by consuming the less time and efforts. During the development of the software customers requirements are volatile in nature and they are changing during the time. By making any change in software, retesting of the software is required to assure that the changes introduced in the software does not put any impact on the other part of the software. It may be possible to add the new test cases to test the modified part of the software. It is very hard to find the affected part of the software and to select the test cases to execute the affected part of the software.

By concentrating on the difficulty possessed by the testing of the object oriented software, prioritization of test cases should be performed to detect the maximum faults which helps to reduce the testing time and cost. To prioritize the test cases some factors are also required on the basis of which the selection and prioritization of the test cases are performed. In this thesis, the test case prioritization techniques for the object oriented software have been presented at the three levels. These levels are the Unit and Integration testing, System testing and Regression testing.

At Unit and Integration level the four test case prioritization technique have been presented. The first technique prioritized the test cases on the basis of the cost and code covered by the test cases. To determine the cost some factors have been considered, which increase the cost in terms of the execution time and space. The second technique prioritized the test cases on the basis of the structural analysis of the object oriented software. The third technique prioritized the test cases on the basis of method complexity. Some factors are considered to determine the complexity of the method. The fourth technique prioritized the test cases on the basis of the analysis of the coupling existing in the software. For the experimental verification and validation all the four techniques have been applied on the software that are implemented in object oriented programming paradigm.

At system level testing the whole software needs to be tested at various grounds like load testing, stress testing, performance testing, etc. Resultant system testing has the large numbers of the test cases. A multilevel test case prioritization for the system testing of the object oriented software has been presented. In the presented technique firstly the requirements are prioritized using the seven factors that are related to requirement. After prioritization of the requirements the modules of the prioritized requirements are prioritized using the four factors followed by the prioritizations of test cases using six factors of the highest prioritized module. Similarly to reduce the testing cost a cost reduction framework (CORFOOS) for the object oriented software has been presented.

At regression testing level three techniques have been presented. The first technique determined the affected paths in software by incorporating the changes in the software and selects the test cases corresponding to the determined paths. In the second level a hierarchical regression test case prioritization for the object oriented software has been presented. The presented technique firstly prioritized the classes on the basis of the testing effort. After prioritizing the classes the test cases are prioritized on the basis of the faults covered by the test cases in the past history. The third technique prioritizes the regression test cases on the basis of some factors related to the past testing history and coverage of the code in term of classes of the software which is going to be retested after incorporating some modifications in it. All the techniques have been validated by applying it on the software.

This thesis focuses on the difficulties of the testing at the Unit & Integration level, System level and Regression testing for the object oriented software. To remove all the difficulties related to testing test case prioritization techniques for each level has been designed. The presented techniques help to deliver the quality software by consuming the less cost with in allocated time. For the applicability of the proposed techniques these have been experimentally validated by applying them on the software implemented in C++ and JAVA. The techniques have been compared with the other existed similar existing techniques. The result shows the efficacy of the proposed work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBRIVATIONS

OOT	Object Oriented Technology
OOS	Object Oriented Software
APFD	Average Percentage of Faults Detection
OPDG	Object Program Dependency Graph
OO	Object Oriented
F	Fault
SF	Sum of Factors
TFC	Total Factor Coverage
TF	Total of Factors
OOCFG	Object Oriented Control Flow Graph
TCPW	Test Case Prioritization Weight
MCG	Method Call Graph
VM	Volume of Method
DM	Difficulty in Method
CM	Complexity of Method
PPV	Path Prioritization Value
FM	Factors in Method
Cinhr	Inheritance Coupling
Cintr	Interaction Coupling
Ccomp	Component Coupling
Vcomb	Coupling Value of combinations of Classes
Cn	Content Coupling
Co	Common Coupling
Ex	External Coupling
Con	Control Coupling
St	Stamp Coupling
Dt	Data Coupling
MC	Modification Coupling
RC	Refinement Coupling
EC	Extension coupling
HC	Hidden Coupling

SC	Scattered Coupling
SPC	Specified Coupling
RDV	Requirement Dependency Value
MDV	Module Dependency Value
RCM	Requirement Coverage Value
FCT	Factor Coverage Value
RPV	Requirement Prioritize Value
MPV	Module Prioritize Value
TC	Test Case
TCPV	Test Case Prioritization Prioritize Value
IRDV	Intermediate Requirement Dependency Value
TE	Testing Effort
CORFOOS	Cost Reduction Framework for Object Oriented Software
RTCPV	Regression Test Case Prioritization Prioritize Value
CDB	Capability of Detecting Bug
CC	Coverage of Classes
ET	Execution Time
CHS	Class Hierarchy Subgraph
RTS	Regression Test Selection
OOPS	Object Oriented Programming System

Chapter I

INTRODUCTION

1.1 OBJECT ORIENTED SOFTWARE TESTING

In past three decades the utilization of the Object Oriented Technology (OOT) to develop the software has widely increased. OOT provides quality software by using its promising features. The OOT supports the features like data abstraction, information hiding, inheritance, polymorphism etc. The problem is represented and understood in natural way by using the OOT. The process of the development of the software using the OOT is different from the other programming paradigms.

To ensure the quality of the software, testing of the developed software is required. Testing in specialized environments requires more attention with the more specialized testing techniques. The testing techniques are dependent on the environment and they may change their working behavior according to the environment. It is very challenging to test the object oriented software as compared to the procedure oriented software. The testing strategies and technology are different for the object oriented software. Most of the testing concepts lose their meaning in OOT.

1.2 TEST CASE PRIORITIZATION

To perform the effective testing within time and budgets the test cases are ordered and executed in such a way that they detect the maximum faults as earlier as possible which helps to deliver the software within specified time and budgets. Some test case prioritization techniques are required to reorder the test cases.

Test case prioritization [1] technique schedules the execution of test cases in an order that attempts to increase their effectiveness in meeting some performance goal. Test case prioritization techniques mainly order test cases according to some criteria that aim to increase the rate of fault detection or maximize the code coverage. Prioritization of the test cases can be done at three levels

- **Prioritization for Regression Test Suite:** In this level the test suite of regression testing is prioritized.

- **Prioritization for Unit and Integration Testing Test Suite:** In this level the test suite of unit and integration testing is prioritized.

- **Prioritization for System Test Suite:** In this level the test suite of system testing is prioritized. Various factors are used to prioritize the test cases of the system testing.

1.3 MOTIVATION AND RESEARCH OBJECTIVES

Object oriented software is different in many ways as compared to the procedure oriented software. Object oriented software is easy to design but testing of the object oriented software is difficult. Most of the testing concepts are meaningless in testing of object oriented software. Object oriented testing techniques and strategies are different from the procedure oriented software. A lot of work has been published to prioritize the test cases for the object oriented software. The researcher proposed various test case prioritization techniques to prioritize test cases for the regression testing and system testing of the software. They used various factors to prioritize the test cases, like coverage based customer priority etc. However there should be some program structure related factors which may be used to prioritize the test cases with the goals to detect the maximum faults as earlier as possible and reliable software. Similarly the complexities of the method and various factors that are contributed to introduce in software have not been addressed in the past literature. In object oriented software there are various concepts if they are not used in efficient way they may become reason of severe faults in software which also not considered to prioritized the test cases. The existence of higher coupling in software makes it very difficult to maintain and test the software. So coupling factor may also be considered for test case prioritization of object oriented software.

The researchers used various algorithms like ant colony, hill climbing etc. Some researchers have taken various factors related to the past history of testing to order the test cases. But they don’t use the efficiency and capability of a particular factor to detect the critical and maximum bug as earlier as possible. Inheritance makes the subclasses dependent on the super class and a change in the super class will directly affect the subclasses that are inherited from it i.e. All subclasses need to be retest. Hence it increases dependency among classes which results in low testability. So, in this case, it is better to check the control flow in the form of classes first and then prioritize the highly affected class and then its test cases. Every test case has the capability to detect the faults whether it is a new fault or detected earlier. A lot of constraints have been imposed on the software industry which may be affecting the quality of the software. These constraints are the budget, time, resources etc. To utilize the limited resources (viz. cost, time, test tools, man power) in an efficient manner, test cases should be reduced and prioritized by identifying the affected paths and affected functions due to modifications in the object oriented systems.

The objective of this research is to design the efficient test case prioritization technique for the object oriented software which helps to perform the effective testing within less time and cost. To achieve this objective, the work on following goals has been performed.

(1) To design and validate a technique for identifying the major changes in software and then prioritize the test cases to test the affected part of software.

(2) To design a test case prioritization technique by determining the most effective factors that contribute in identifying the highly important test cases such that there will be high rate of fault detection for an object oriented software system.

(3) To design and validate a technique for system test case Prioritization for object oriented software based on types of requirements, complexities included in requirement modules complexities, fault proneness etc.

1.4 CHALLENGES OF TEST CASE PRIORITIZATION

1.4.1 Selection of Test Case and their Prioritization during Regression Testing of Object Oriented software: Selective retesting of the software is performed to identify that modification in software has not caused unintended effects in software. Modifications can be done by adding or deleting a class, interface or a function. It is very challenging task to identify affected part of the software by modification and to select test cases corresponding to the affected paths. The size of the test suite may grow as the software gets modified. It is very expensive and time consuming to execute all the test cases.

Solutions: *To solve the problem a regression test selection technique for object oriented software is proposed. The proposed approach used the Object Program Dependency Graph (OPDG) and dynamic slice to determine the affected path and select the test cases. To cope with the issue of identifying of affected part in the modified software, two test case prioritization techniques for object oriented software are presented. The first technique works at two levels. At the first level classes are prioritized on the basis of the calculated testing efforts. At the second level test cases of the prioritized classes are prioritized using the past execution of the test case.*
In the second technique some critical factors are considered. Every factor has been assigned a positive weight which shows the criticality of the factor. By using this factor the regression test cases prioritization technique for object oriented software is presented.

1.4.2 Test case prioritization for Unit and Integration testing of object oriented software: A lot of time, effort and cost is spent to perform the unit and integration testing as a large number of test cases are needed to be executed. So there should be an efficient technique to prioritize the test cases.

Solutions: *To cope with this issue, five test case prioritization techniques for object oriented software are proposed. These proposed techniques considered some factors for prioritization of the test cases. The factors are considered on the basis of their capability to detect the maximum faults by less efforts or having the higher chances to introduce faults in the software.*

1.4.3 Prioritization of Test Cases for System Testing and Designing a framework to reduce the testing cost: In system testing the software is need to be tested in the real conditions which are very challenging. So large numbers of test cases are generated in object oriented environment.

Solutions: *To resolve the above challenge a technique to prioritize the system test cases for object oriented is presented. The presented approach works at three levels. At fist level requirements are prioritized using the seven factors. At the second level the modules are prioritized using four factors. At the third level the test cases of prioritized module are prioritized using the six factors.*

A framework to reduce the testing cost to test the object oriented software is also presented. The presented framework prioritizes the requirements which are going to test in three categories. Further the categorized requirements are mapped with the past testing history of the software tested by the industry. After this testing strategies are decided which help to deliver the quality product within the lowest testing cost and time.

1.5 ORGANIZATION OF THESIS

The thesis has been organized in the following chapters:

Chapter 1: Covers the introduction of the thesis.

Chapter 2: The basic concepts of object oriented software testing, regression testing and test case prioritization are discussed in this chapter. A detailed review of the available test case prioritization techniques for object oriented software and the issues associated with these techniques are also discussed.

Chapter 3: In this chapter four test case prioritization techniques for unit and integration testing for object oriented software are presented in this chapter. In first technique, the test cases are prioritized on the basis of cost and code covered by the test cases. The second approach prioritizes the test cases on the basis of structural analysis of the object oriented software. The third technique prioritizes the test cases on the basis of the method complexity. The fourth technique uses the existing coupling in the software to prioritize the test cases. The proposed techniques have been validated by applying it on software. To show the effectiveness of the proposed techniques the experimented results are compared with existing similar techniques and non prioritized approach.

Chapter 4: This chapter covers a multilevel system test case prioritization technique for prioritizing the system test cases of object oriented system. It also describes framework for reducing the testing cost for object oriented system. The efficiency of the proposed technique is evaluated by comparing with non prioritized as well as previous existing approaches.

Chapter 5: This chapter is concerned with prioritization of the test cases while performing regression testing. It is further divided into three sections. In first section, a regression test case selection technique for object oriented software based on OPDG and dynamic slicing is presented. The second section of the chapter discusses a fault severity based technique to prioritize the regression test cases. The third section discusses a history based technique for regression test case prioritization of object oriented software. All the proposed techniques in this chapter have been validated and the results obtained show the efficacy of these techniques.

Chapter 6: It concludes the outcome of the work proposed in this thesis. It also discusses the possibilities of future research work based on the proposed approaches.

Chapter II

LITERATURE REVIEW

2.1 INTRODUCTION

In last two decades most of the software has been implemented using the object oriented programming system. The software industry has adopted object oriented technology (OOT) to develop the software. In almost every field the OOT is a preferred programming paradigm. These fields may be artificial intelligence, graphics, exploratory programming physics, telecommunications etc. Object oriented technology has various features that make it popular. These factors are the data abstraction, information hiding, extensional programming and reusability of the code.

The process of developing the software using the object oriented technology is different than the developing of the software using the procedural oriented programming system. The software developed using the procedural oriented technology has higher complexity as compared to the software developed using the object oriented technology. The complexity of the software is main reason to adopt the object oriented technology. Besides the merits of the object oriented technology testing of the object oriented software is very challenging. Lots of testing concept of the procedural oriented language has lost their existence in object oriented technology. The design of the software is very easy but it is hard to test and maintain as compared to the procedural oriented software. Object oriented software has [2] some testing and maintenance problems listed below

- **Understanding Problem** In object oriented software invocations of many functions take place due to the information hiding and encapsulation. It is very difficult to understand the sequence of the execution of the functions and design the test cases corresponding to the identified sequences.

- **Dependency Problem** In object oriented technology there is tightly coupled relationship between the inheritances, aggregation, association, class nesting, function invocations, polymorphism etc. These relationships show how the one class depends on another class. The complexity of the relationship increases the difficulty for testing and maintenance of the object oriented software.

- **State Behavior problem**: In object oriented technology every object has state and state dependent behaviour. Object changes its state when any operation is applied on the object and the combined effect of the applied operations on the objects should be tested.

2.2 IMPACT OF OBJECT ORIENTED TECHNOLOGY ON TESTING

Software development organizations are use the object oriented technology to [3] enhance the productivity and efficiency of the software. To assure higher productivity and efficiency, more testing efforts are required to test the software. There are many factors [4] which distinguish the OOT from the procedural oriented technology. These factors are the encapsulation, data hiding, inheritance, reuse and abstraction.

- **Encapsulation:** In encapsulation one or more elements are bounded in a single container. Encapsulations have three levels, low – level,[5] mid- level and high level. The low level contains the array and records, mid level contains elements like subprograms and subroutines and high level contains the items like classes, packages and objects. In object oriented testing the basic unit of the testing is the object or classes. In OOT the functions are allied with the object and state of the object defines the behavior of the object. Berard [5] introduces two types of impacts of encapsulation on testing of object oriented software.

    1. The change in the definition of the Unit
    2. Impacts of change in the definition of unit on the integration testing

- **Information Hiding:** Information hiding is the hiding the object details that do not contribute to its essential [6] characteristics. It hides the structure of the objects as well as the method implementations. The concepts of the information hiding make the testing very challenging. If the tester wants to test the method the access of the internal state of the object or data is required, which is hidden from the tester.

- **Abstraction:** Abstraction focuses [6] on the outside view of the objects. It shows only the essential behavior of the objects and hides its implementations. To test the object, information abstracted by the object is required but it gives only black box view of the object.

- **Inheritance:** In Inheritance one class shares its structure or behavior with the one or more classes. The implementation of inheritance means derived class acquires all the properties of the base class. In inheritance functions can be redefined or override in the derived class. For executing the different member functions, concept that can execute the different functions based on pointer type of the object must be considered by the testing techniques. The inherited features of the base class also require the retesting when these are inherited in the derived class. The testing of the derived class is affected by the retesting of all the features of the base class. The issues in testing of inheritance are given below

    - Superclass modification
    - Inherited methods
    - Reusing of test suite of superclass
    - Addition of subclass method
    - Change to an abstract superclass interface
    - Interaction among methods

2. 3 REQUIREMENT TESTING

The cost of the removal of the bug is directly dependent on the creation and the detection of the bug. The cost of fixing the bug at later stage is more as compared to the bug detection at earlier stages. Testing of the requirements helps to reduce the testing cost by detecting the bug at earlier stage. Requirement testing also helps to deliver the software in specified time. The requirements are delivered to designers by analysis of the requirement of the customers.. The result of analysis of the requirements is the description of functions that are performed by the system. The commonly tools used for the requirements analysis are Prototyping, Graphical User Interface, Requirements specification model, Domain object model and Use case.

The testing of requirements is performed to validate the quality of the output of requirement analysis phase and detect the maximum errors at earlier phase. The Requirement testing has three basic issues namely Correctness, Completeness and consistency.

2.4 DESIGN TESTING

The testing of design is performed to assure whether the design will meet the required specification or not. It is very costly to fix the bugs at later stage. So it is imperative to test the design by utilizing the best resources of the organization. The three category of the design has been given by the D. Champeaux. These are [7] the functional, physical and performance design. All three categories are focused on the goal. The software design phase uses the functional requirements, resource requirements and performance requirements from the requirements analysis phase. The class diagrams and object interaction diagrams are used to describe the architecture of the object oriented system. There are five objectives to perform the testing of design of the object oriented system. These are the consistency, completeness, feasibility, correctness and traceability.

2.5 BASE CLASSES UNIT TESTING

Class is a basic unit for the development and testing of the object oriented software. Unit testing contains the verification of the smallest part of the software which is going to develop. It gives the assurance that individual parts of the complex system work according to their specifications. Some motivations and objectives behind the testing of a class are given below [8] , Completeness, Early Testing, Easy Debugging , Better Coverage, Better Regression Testing, Reduced future testing effort, Better quality system . In Unit testing a class should be tested at the following three [9] levels.

2.5.1 Functional Testing of Methods: In this, every method of the class should be tested according to their functions and does not consider its implementation. Every method should be tested individually.

2.5.2 Structural Testing of Methods: In structural testing of the methods it should be tested in such a way that all the feasible paths must be covered by the design test cases.

2.5.3 Interaction Testing of the Method: After testing the method in isolation, interaction of method with the other methods should be tested. The working of the methods also depends on the other methods which are associated with it.

2.6 DERIVED CLASSES UNIT TESTING

Inheritance of classes is the basic feature of the object oriented programming system. In inheritance the various classes are related in the hierarchical relationship and share the common features between the different classes. Every class is created to implement some functionality and classes are logically related to the other classes. Inheritance is used to bind the two classes in logical way. In inheritance the two classes are logically related to each other and a class is derived from the other class called base. The derived class acquires all the properties (data member, member's functions) of the base class.

There are two ways to test the derived class, which are given below

(a) Test a derived class as a flattened class. In this all the inherited attribute of the base class should be tested in the derived class.

(b) If the base class is already tested then the inherited attributes of the base class need not to be retested in the derived class

2.7 TESTING SPECIAL FEATURES OF OOS

In this Section some special [4] features for the object oriented software are discussed.

2.7.1 Static Data members: A static member is shared by all the objects of the class. Static members are used not only the part of the object of the class but it can also be used without creating the object of the class. A special testing technique is required to test the static data members. In a class it is possible to have the static data members and static member functions together. Static member's functions can directly use only the static data members of class. For testing the static data member static slice is used.

A static slice is slice of the class whose data members are declared as static. Two types of testing is required by static slice

(1) Testing as a part of class
(2) Testing as Stand- Alone data members

2.7.2 Function Pointers: Functions pointers are used to change the behavior of functions at run time. A function pointer points to the address of the functions. It is just like the ordinary pointer. It is also initializable, modifiable, and reportable like the other data members. A slice is created for the function pointer and all the members' functions that manipulate it. The functions pointer is tested for all the possible combinations.

2.7. 3 Structure as Data members: Structure is used to combine the different type of the data items. It can be declared as a data member of the class like the other data members of class. The state of the object of class depends on the state of the structure type data member of the class. A modification in the state of the structure puts impact on the state of the object of class. For such type of class multilevel testing technique is required.

In the first level of the testing the reference of a structure is tested. Only one slice of this structure type is used and considers the modifications in its references. In the second level of testing the slice is further subdivided in subslices. Every subslice represents the unique elements of the structures.

2.7.4 Nested Classes: In nested classes a class is declared inside the other class. Nested class is used to minimize the number of the global names. Nested class can be used for the following objectives

- To resolve the naming issues
- For containment purpose

In case of naming issues the nested class can be tested as a separate class without any special type of testing. In the case of containment, purpose multilevel testing technique is required. The suggested three level testing techniques is given below

- Testing a Pointer/Reference as a data members
- Testing a nested class as a standalone class
- Testing a Nested Class in the scope of the enclosing class

2.7.5 Members Access Control: A class can control accessibility of its members to functions other than its own member functions [10]. Member functions of class acquire access either by default or by the use of the public, private and protected.

Private: private members are only used by the member functions and friend function in the same class.

Protected: Protected members are used by the member functions and friend function of the same class and any derived class form the class in which they are declared.

Public: Public members can be used by any function.

Three levels of testing technique is used to test the three type of members access controls

- Testing a class from unrelated client's perspective
- Testing a class from derived class's perspective
- Testing a class from its own perspective

2.7.6 Composite Classes: A Composite class can be created by two or more objects. The objects that are used to create the composite class are known as the composing objects. The two level testing techniques are required to test the composite classes.

- Testing a Pointer /Reference
- Testing a Composite class with composing classes

2.7.7 Abstract Class: Abstract class is used to give the common interface for the different types of the derived class. There is no object of the abstract class but pointer to object of type of abstract class can be declared. Since the object of the class is not created so testing of the abstract class is not required but to minimize the testing of derived class of the abstract class minimal testing should be performed.

2.8 INTEGRATION TESTING

Integration testing is testing approach to detect the errors when two or more individual developed components are combined together with objective to fulfill the required functionalities. Generally the errors related to the integration testing are the interface errors, timing errors and throughput errors. In the object oriented software every component has a state and integration may affect its behaviors. For performing the integration testing, three types of the testing techniques are there, Execution based integration testing, Value based integration testing, and Function based integration testing.

There are the following possible combinations of the integration.

- Integration of members into a single class
- Integration of two or more classes using inheritance
- Integration of two or more classes using containment
- Integration of two or more classes to build a component
- Integration of many components to develop an application

The main objective of the integration testing is to assure that all the individual components are combined and obtained the desired goal without errors or any failure. Bill hetzel [10] determined the five considerations for planning of integration testing. These five considerations can be summarized in the following questions

- How many objects should be assembled before integration testing?
- What should be the order of the integration testing?
- Should be there more than one skeleton for integration?

2.9 INTEGRATED SYSTEM TESTING

In the system testing whole system is tested to assure that whether the developed system meets the desired functionality or not. The system testing includes the integrated system testing, alpha testing, beta testing, and the user acceptance testing. During the system testing, all the functional test, performance test , stress test and the resource requirement test should be performed. Many systems are unable to perform correct functionality during the performance and stress testing of the software. Sometimes the software is not able to perform in real working environment. Performance testing of the system should be tested according to its performance requirements. The heavy volume of data should be used to test scalability of the system. The occurrence of the deadlock and termination should be checked by operating the system for hours, days and months. The system should be executed on the different types of hardware and software platforms to verify its portability. Software specification documents must be checked for correctness, consistency and completeness of the system.

In system testing the system should be tested for all the possible combinations of the data for all conditions. But it is not feasible due to time and resource constraints. All the requirements should be exercised and select the significant data that covers the broad range of the usage. The following types of testing should be performed for system testing

- Sanity testing
- Functional Testing
- Human factors testing
- Performance testing
- Capacity /load testing
- Documentation testing

2.10 LEVELS OF TESTING FOR OBJECT ORIENTED TESTING

There are four levels [11] for the testing of the object oriented software. The number of testing levels depends on the testing approach. Generally the object oriented testing is done at four levels. These levels are

- Method Level Testing
- Class Level Testing
- Inter class Testing (Cluster Level Testing)
- System Level Testing

Method Level Testing: In method level, testing of an individual method is performed. The methods of the class are tested by applying the techniques used for the conventional programming language.

Class Level Testing: The data members and the member functions are combined in a class. The interaction among the different functions of an individual class is tested. The testing of an independent function is challenging**.**

Inter Class Testing: The classes in object oriented software are interacting through objects and parameter passing. Inter class test is performed to test the interaction among the different classes.

System Level Testing: The cluster of the classes makes the system. In system level testing whole the system is tested at various levels. System level testing is concerned with the input supplied by the user and output visible to the outside user.

2.11 OBJECT ORIENTED TESTING TECHNIQUES

In this section three [12] popular techniques for object oriented software are discussed. These techniques are the path based testing, state based testing and class testing.

2.11.1 Path Based Testing: In path based testing the source code is converted in the activity diagram. The activity diagram shows all the sequence of the activities performed by the source code. The Unified modeling language (UML) is used to create the activity diagram. The activity diagram shows the basic and all possible alternate flow of the software. Every activity is represented by the rectangle with round corner and transition between the activities is represented by the arrow.
The activity diagram provides the basis of the path testing where all independent paths are determined and are executed at least once.

2.11.2 State Based Testing: In state based testing for the object oriented software a state machine is used. In the state machine the output of state machine does not depend only on the present state but also on the past state. The model of the behavior of the objects is created by using the state machine. Every state corresponds to the certain value of the attributes and transitions of the methods. It is expected that the states are visited by the objects during its life time in response of events. The state machine is represented by the state chart diagram which is created by using the UML. The state machine shows the flow of one state to another state. The states are denoted by the rectangle with round corner and transition between the states is shown by the arrows. Two special states named alpha and omega are used to represent the constructor and destructor of the class.

2.11.3 Class Level Testing: Class is the basic testing unit of the object oriented software. The testing of the class is performed to verify the implementation according to its specification. Class testing is like the unit testing of the conventional testing for the object oriented software. The class cannot be tested in isolation. It requires additional code for testing. The test cases are designed to test the test cases, a test driver is required to execute each test cases. One or more instances of the class is created by the test driver to execute the test cases.

2.12 BUG CLASSIFICATION BASED ON CRITICALITY

Bugs are classified on the basis [2] of the impact on the software which is under testing. The bugs are classified in the four categories on the basis of their criticality.

Critical Bugs: These types of bugs stop the functioning of the software. The user is not able to operate the software.

Major Bugs: These types of the bugs do not stop the functioning of the software but does not give the results as per its desired results.

Medium Bugs: These types of the bugs cause the output not according to its standard or conventions.

Minor bugs: These types of bugs do not put the impact on the functionality of the software.

2.13 OBJECT ORIENTED DESIGN PRINCIPLE

In this section [12] principles of object oriented design are presented

- **Single Responsibility Principle:** A class should be designed only for a single responsibility because each responsibility is a cause of changes in a class. The classes become large and complex if many responsibilities are handled by a single class. For avoiding this situation, it is mandatory to ensure that the code is simple.

- **Open Closed Principle:** Software entities like classes and modules should be designed in such a way that they are open for extension and closed for modification. All new functionality should be added in the code by adding a subclass to the existing class without making any change in existing classes.

- **Liskov Substitution Principle**: The instance of super class is replaced by the instance of the derived class. If this is not followed, the class hierarchies become messy.

- **Interface Segregation Principle**: The class should depend on the smallest possible interface.

- **Dependency Inversion Principle**: Modules that implement the high level policy should be dependent on a well-defined interface rather than on modules that implement low level polices.

- **Principle of Package Cohesion**: If the classes are changed or reused at the same time, only then they should be grouped together, otherwise they should not be grouped together.

2.14 COUPLING IN OBJECT ORIENTED SOFTWARE

Stevens et.al defines [13] coupling as the measure of the strength of association established by one module to another module. Module having the strong coupling with other module is difficult to understand and modify to correct its working. Strong coupling existed between the modules increases the complexity. The complexity of a software can be decreased by designing the system is such way that lowest coupling exist between the modules. For object oriented system [14] the following types of couplings are deduced

(a) **Interaction Coupling:** In interaction coupling the methods are called by each other and data is shared by the method. If any class is having the highest coupling of methods within the class, that means class is very complex. The interaction coupling is further subdivided in coupling dimensions as given below:

- **Content Coupling:** In content coupling, one method can access the directly or indirectly the implementations of the other method.

- **Common Coupling:** In common coupling, methods are coupled through unstructured and global shared data.

- **External Coupling**: In External Coupling two methods of the same class use the same variable which is acting as a global variable in module.

- **Control Coupling :** In control coupling, one method controls the internal implementations or logic of the other methods

- **Stamp Coupling:** In stamp coupling, one method passes the whole data structures as parameter to the other method.

- **Data Coupling:** In data coupling, two methods are communicated through the parameter only.

(b) Component Coupling: In component coupling, one class is used as a domain by any instance variable of the class. The component coupling is further subdivided in coupling dimensions as given below:

- **Hidden Coupling:** The coupling between the two classes C1 and C2 is said to be hidden if the object of C2 used the implementation of a method of C1 whereas C2 is not shown in the specification and in the implementation of C1.

- **Scattered Coupling:** Two classes C1 and C2 are scattered coupled if any local variable or instance variable of C1 uses C2 as domain.

- **Specified Coupling:** In specified coupling, one class is included in the specification of the other class.

(c) Inheritance Coupling: In inheritance coupling, one class is directly or indirectly subclass of another class. The inheritance coupling is further subdivided in coupling dimensions as given below:

- **Modification Coupling:** In Modification coupling, the inherited information of the super class is changed by the subclass.

- **Refinement Coupling:** In Refinement coupling, the subclass adds some new information to the inherited information and changes only due to predefined rules.

- **Extension Coupling:** In Extension coupling, subclass adds some methods or variable without changing the inherited information from the super class.

2.15 PROGRAM SLICING

Program slice was presented by the Weiser for debugging of a program [15]. Program slicing is execution of the set of statements of a program. A slicing criterion is used to create slice of a program. Slicing criterion is a point in program where the computed value is impacted by the set of statements. Slicing criterion is a pair (S,V) the statement S in the program and a variable V in the statements S. The set of statements of a program which have a direct and indirect impact on the computed value at slicing criterion is called a program slice with respect to slicing criterion.

There are following types of slicing techniques:

- **Static Slicing:** Static slicing is a set of statements of a program that may put impact on the value of variable of a particular statement for all possible inputs. The backtracking dependencies between the statements are used to compute the static slicing.

- **Dynamic Slicing:** Dynamic slicing is the set of statements that may put impact on value of variable for specific set of inputs rather than for all inputs. For dynamic slicing specific information of a program execution is used.

- **Backward Slicing:** Backward slicing contains the set of statements of program that may impact the slicing criterion directly or indirectly.

- **Forward Slicing:** Forward slicing consists of the statements of a program which may be impacted by a variable V at the particular point which is used and defined.

2.16 TEST CASE PRIORITIZATION

The size of test suite increases as the software evolves. Due to time, resource and budget constraints, it is imperative to prioritize the execution of test cases so as to increase the possibility of early detection of faults. Test case prioritization technique has become very effective technique to detect the faults as earlier as possible. Prioritization of test cases can be performed at various stages like potential of fault detection, statement coverage and branch coverage. Due to large functionality of the software there is large test suite to test the software. It is not necessary that every test case incurs a fault. For executing all the test cases testing team requires more resources and time thereby increasing the cost of the testing. Hence due to testing the project may go out of budget or may get delayed. The order of test cases also affects the process of testing and it also helps in reducing the cost of testing of project. As the cost to fix the bug in early stages incurs less cost as compared to fix the bug at later stages. It may be possible that earlier test cases report the entire bugs that are also reported by the test cases which are executed later.

2.17 REGRESSION TESTING

Regression testing is the process to ensure that modified software is working according to the required specification and the modified part of the software has not put any affect on the unchanged parts of software [16]. Studies show that regression testing accounts for 80% of the testing costs [17]. Shifts in software development practices towards component based software development and agile development impose constraints on regression testing [18], giving rise to approaches that minimize the cost of regression testing.

In Regression testing a set of test cases is selected from existing test suites to verify that changes made in software have no unintended side-effects [19]. It is very challenging task because many software has large test suite and changes in software are incorporated rapidly. To make the regression testing more effective and efficient various regression testing techniques have been developed, but many problem remain, such as Unpredictable performance, Incompatible process assumptions and inappropriate evaluation models.

2.18 AVERAGE PERCENTAGE OF FAULTS DETECTED (APFD)

Elabus. et al. [20] presented an APFD metric to measure the weighted average of the percentage of detected faults by execution of the test suite. The value of the APFD is in range of 0-100, where higher APFD value shows the higher detection rate of the faults. APFD is calculated by the formula given below

APFD = $1-((TF_1 + TF_2 + TF_3 + \text{-------------------} TF_m)/nm) + 1/2n$

Where TF_i is the position of test case in the test suite T that detects the fault i

m is total number of faults detected by test suite

n is the number of total test cases in the test suite T

2.19 TESTING OF OBJECT ORIENTED SOFTWARE USING COUPLING

In this section a review of various research papers related to testing of object oriented software using coupling is presented.

Varun Gupta et al. [21] proposed a coupling metric for measurement of package level coupling. The proposed metric considers the different type of connections between different packages. These connections are class- class, sub package – sub package, sub package – class and class – sub package. They also considered the hierarchical structure of package and direction of connection between packages.

Vipin Sexena et al. [22] discussed the impact of coupling and cohesion in object oriented software. They used metric DCH (degree of cohesion) by exploring two another metrics MRC (message received coupling) and DCP (degree of coupling). This coupling mechanism helps to measure the functional strength of class of an object oriented system.

A new technique [23] for analyzing and testing the polymorphic relationship in the object oriented software presented by Roger T Alexander and Jeff Offutt. They summarized new testing criteria to address problems that arise from inheritance and polymorphism. The couplings have been updated and applied to the object oriented software to handle the aggregation inheritance and polymorphism. The foundation of proposed technique is coupling sequence.

Eric Arisholm et al. [24] presented the measurement of coupling by dynamic analysis of systems. They presented formal operational definition for measures of coupling. They also described a tool for collecting such measures from Java programs effectively.

Varun Gupta et al. [25] introduced dynamic cohesion metrics. The metrics provide the scope for measurement of cohesion up to class level. The experimental validation found that dynamic cohesion metrics are more accurate and useful.

The measurement of coupling [26] which is based on the object oriented relationship between the classes of the object oriented software is presented by Jeff Offut et al.. They concentrated on the type of coupling which are unavailable after the software has been developed. The coupling is divided in four types. They also presented a static tool which is used to determine the coupling between the classes of java packages.

V. S. Bidve et al. [27] presented the coupling metric for object oriented design. They used the specially adapted software metrics to investigate the run time behavior of the objects in Java programs. The considered metric quantifies the coupling at levels of class to class and object to class. For every measurement they indicate the use of

coupling type, factors used to identify the coupling strength, indirect coupling accounting when coupling are imported and exported.
Roger T Alexander et al. [28] presented an approach for analyzing and testing the polymorphic relationship of object oriented software. They summarized the data flow testing technique and new testing criteria that are used to isolate the problem that occurs due to use of inheritance and polymorphism.

James M. Bieman et.al. [29] presented the evaluations of the effectiveness of the criteria for detecting the faults that are outcome of the polymorphic relationship existed in object oriented software. The performed experimented evaluated the three coupling based test criteria for integration testing. These criteria are all coupling sequences, all– polly classes and all poly-coupling - defs-uses. The experiment result shows that the technique is effective testing strategy for object oriented software that uses the inheritance and polymorphism.

A technique [30] to reduce the coupling existed in the object oriented software is presented. The presented algorithm has four phases. These phases are authentication, selection of two object oriented files, count the number of classes/object/inheritance and deduction of better approach in current situation.

Zhenvi Jin et al. [31] proposed a coupling based integration testing technique. They defined four coupling based criteria, call – coupling, all- coupling - defs, all-coupling-uses and all – coupling- paths. The proposed technique has been compared with the category-partition method and inters procedural data flow testing method. The outcome of the comparison shows that the proposed technique detects more faults with the fewer test cases as comparisons with the other methods.

To generate the test cases [32] for object oriented integration testing coupling relation of unit is used. The technique considered the DU pairs for selecting the method sequence which are further used to generate the test cases.

An algorithm [33] to solve the class integration test order (CITO) problem is presented. The findings include superior edge weight. The weights are derived from quantity coupling measures. The weights are used on nodes resultant allowing more information to be used. For validation of the proposed technique it was compared with the other technique.

Michela Pedroni et al. [34] analyzed the dependency structure of the object oriented concept. By an analysis of the dependency structure, they found that basic object oriented concepts are tightly interrelated.

2.20 TESTING OF OBJECT ORIENTED SOFTWARE USING INHERITANCE

In this section a review of various research papers related to testing of object oriented software using inheritance is presented

Sujata Khatri et al. [35] presented the analysis of some factor which affects the testing of object oriented system. These factors are Data abstraction, inheritance, polymorphism, coupling, cohesion among methods and abstract classes. These factors introduced new challenges in testing for object oriented system.

Muhammad Rabee Saheen et al. [36] presented a how cost of unit testing is predicted using depth of inheritance. They relate the depth of inheritance tree (DIT) with respect to number of methods to test in each class. In this paper they also distinguished two types of testing strategies and two types of inheritance tree.

The UML design based metric has been presented by the [37] Gagandeep Makkar et al. The proposed metric considered the number of inherited attribute and depth level of class. They also considered the penalty factor. If the reusability decreases then the penalty factor increases.

Nasib S. Gill et al. [38] characterized metric of reuse and reusability in object oriented software development. They presented five new metrics. The proposed new metrics are breadth of inheritance tree (BIT), Method reuse per inheritance relation (MRPIR), Method reuse per inheritance relation (ARPIR), generality of classes (GC), and reuse probability (RP).

An empirical investigation [39] into the modifiability and understandability of object-oriented (OO) software is presented by R. Harrison et al. They conducted a controlled experiment to establish the affects of various levels of inheritance on modifiability and understandability. The results indicated that the systems without inheritance were easier to modify and understand than the systems containing three or five levels of inheritance.

John Daly et al. [40] performed experiment and collected data to test the effect of inheritance depth on maintainability of object oriented software. The collected data showed that maintaining task for the object oriented software with the three levels of inheritance depth is quicker than maintaining the equivalent object oriented software with no inheritance.

Arti Chhikara et al. [41] presented an assessment of effect of the inheritance on the object oriented Systems. Their assessment showed that inheritance is a key factor of object oriented Systems.

Mary Jean harrold et al. [42] presented an incremental class testing technique that uses the hierarchical nature of inheritance relations among classes. Base classes are tested first by designing a test suite that tests each member function individually and also tests the interactions among member functions. In order to design test suite for subclasses, a subclass have to inherit testing history from its parent class. A testing history guides the execution of test cases since it indicates which test cases must run to test the subclass. Only the new attributes or affected, inherited attributes are tested and the parent’s class test suites are reused.

Gregory Seront et al. [43] presented the relationship between the degree of object orientation of software entity and cyclomatic complexity. They observed that there is no significance correlation between the depth of inheritance of class and its weighted method complexity.

The object oriented program dependence graph (OPDG) for representation [44] of object oriented programs. The representation is composed of three layers: these layers are Class Hierarchy Subgraph (CHS) , Control Dependency Subgraph (CDS) and

Data Dependence Subgraph (DDS) The presented representation divided in to three layers First layer presents the structure of class inheritance, second layer presents the control dependence and data dependency subgraph with objects and third layer shows the dynamic and runtime aspects of object oriented programs. They also introduced new definitions of definition (def) and use of variable.

2.21 TESTING OF OBJECT ORIENTED SOFTWARE USING SLICING

In this section a review of various research papers related to testing of object oriented software using Program slicing has been presented

Loren Larson et al. [45] presented a system dependence graphs on which slicing can be applied. The system dependence graph constructed for individual classes, groups of interacting classes and complete object oriented program. The presented system dependence graph consists of program dependence graph and class dependence graph. Program dependence graph represents the main program in the system and class dependence graph represents classes in the system. A two pass algorithm is used for computation of slice in system dependence graph.

Anand Krishnaswamy et al. [46] addressed the issues to represent the slicing of object oriented program. For representation of object oriented program the author designed a representation which is based on program dependency graph. The concepts like polymorphism, dynamic binding, class inheritance and message exchange between objects were also represented. second They presented an algorithm that demonstrates the applicability of the object oriented program dependency graph for slicing object oriented is proposed.

2.22 MODEL BASED TESTING OF OBJECT ORIENTED SOFTWARE

In this section a review of various research papers related to testing of object oriented software based on various models is discussed.

David P. Tegarden et al. [47] proposed a model of software complexity for object-oriented systems. In this model there are four levels of software complexity of object-oriented systems: variable, method, object and system. At each level there are measures which account for cohesion and coupling aspects of system at that level. The measures identified are consistent with the characteristics of good OO design.

Santosh Kumar Swain, et al. [48] presented the testing of object oriented software based on a model in which test case derived represents the software behavior. The proposed model based approach carried out at the time of software development for automatic testing of object oriented software.

The object relation diagram model (ORD) is reverse engineering based and constructed by analyzing the C++ source code of an object-oriented program [49]. An ORD is a directed graph in which vertices represent the object classes and edges represent the relationships among object classes. The test order is generated from the ORD by using an algorithm called test order algorithm for unit testing and integration testing of object-oriented programs. This algorithm uses topological sorting and clusters of strongly connected subgraphs of the ORD. An optimal test order is computed such that the effort required to construct the test stubs to simulate the untested classes/ member functions is minimum.

Model based approach [50] increases the flexibility and efficiency of the development as well as quality and reusability of results. Also varieties of test patterns are presented for the design of testable object-oriented systems. The proposed approach uses explicit models for test cases instead of trying to derive test cases from a single model.

Pranshu Gupta et al. [51] applied a class dependency model to object oriented programs. In this paper they created hierarchy of testing order using the class dependency model and analyzed where the faults are concentrated in test order hierarchy. Based on their analysis the author showed that there should be different approach for defining the test order for various categories of faults.

Mahfuzul Huda et al. [52] proposed an effectiveness quantification model of object oriented design. The proposed model uses the technique of multiple linear regressions between the effectiveness factors and metrics. Structural and functional information of object oriented software has been used to validate the assessment of the effectiveness of the factors. The model has been proposed by establishing the correlation between effectiveness and object oriented design constructs. The quantifying ability of model is empirically validated.

Anil Kumar Malviya et al. [53] presented some observation on maintainability estimation model for object oriented software in requirement, design, coding and testing phases. The presented work is about increasing the maintainability factors of the metrics.

Dinesh Kumar Saini et al. [54] analyzed the security issues related with the architectures of the object oriented system and created a model for security assessment. The proposed model is based on risk and it is widely accepted form of security measurement.

An approach [55] for predicting the run time errors was introduced by Bremananth R. The proposed fault prediction model is designed to separate the faulty classes. The separated faulty classes are classified according to the fault occurring in specific class. This approach concerned with faults due to inheritance and violations of java constraints.

2.23 TESTING OF OBJECT ORIENTED SOFTWARE USING METRIC

In this section a review of various research papers related to testing of object oriented software using various metrics has been presented.

parvinder Singh Sandhu et al. [56] proposed dynamic metrics for polymorphism in object oriented system. They addressed some important factors which may impact their usefulness. These factors are Dynamic, Robust, Discriminating, Unambiguous, Platform Independent. They also presented the classification of metrics. The classified category of metrics is Value Metric, Percentile Metric, Bin Metric and Continuous Metric.

Victor R. Basili, et al. [57] analyzed the results of study done at the University of Maryland for the object– oriented design metrics introduced by Chidamber& Kemerer, [174]. To evaluate their results they gathered data about defects found in object –oriented classes. Then by comparing the results of their experiment to this data they made a conclusion that five out of six Chidamber & Kemerer's OO metrics appears to be useful to predict class fault-proneness during the early phases of life cycle. They also concluded that these metrics are better predictors than code metrics.

Seyyed Mohsen Jamaliin et al. [58] identified that software development process engineers are shifting towards the new processes or approaches with most prominent being object-orientation. So to manage the process there is a need for metrics suite for object-orientation. They also presented a basic metric suite for object oriented design.

The influence of program elements metric was proposed by Amarnath Singh et al. The proposed [59] metric is used to find out most critical elements of program In the proposed approach they used the intermediate graph representation of the program. By using the forward slicing on graph with the help of which influence of class is determined that shows the capability of class to cause failure.

Arti Chhikara et al. [60] presented a set of metrics. The presented metrics are used to order the programs based on their complexity values. They concluded that there should be compromise among internal attribute of software to maintain the higher degree of reusability.

Magiel Bruntink et. al[61] analyzed the relation between classes and their JUnit test cases. They demonstrated a significant correlation between the class level metrics and test level metrics. They also discussed how various metrics can contribute to testability. They conducted the experiments using the GQM and MEDEA framework. The results are evaluated using the Spearman's rank order correlation coefficient.

Ravinder Kumar Gupta et al. [62] proposed a testing technique which is based on the state and collaboration models of system. The object interactions are tested by considering state transition of objects and the corresponding activities taking place in use case. They constructed a state collaboration diagram (SCOTEM) and generated the test cases to achieve state activity coverage of SCOTEM.

2.24 TESTING OF OBJECT ORIENTED SOFTWARE USING INTERMEDIATE REPRESENTATION OF THE SOURCE CODE

In this section a review of various research papers related to testing of object oriented software using various representation are presented.

Xiaolan Wang et al. [63] proposed a method for construct a dependency graph of error statement. They applied the symbolic execution and constraint solving to object oriented software exact testing. The presented method can be used in many systems and it is capable to detect the errors in different languages.

An algorithm that directs the construction of functional [64] test cases for a class was introduced by the Juliana Georgieva and Veska Gancheva. In the proposed algorithm test cases are constructed from state representation of the specification of class. The algorithm also provides the basis for automating an increasing amount of the testing process for object oriented system.

Nirmal Kumar Gupta et al. [65] presented a method that uses genetic programming approach for generating test cases for classes in object oriented software. In this method a tree representation of statements in test cases is used. The proposed method strategies for encoding the test cases and using the objective function to evolve them as suitable test case are presented.

A Call – based Object [66] Oriented System Dependence graph for object oriented program gives representation of object oriented program based on dependency. The proposed representation considers the object oriented features like inheritance and polymorphism. They also included method visibility in a derived class and different types of method call edges to describe different calling context.

Ranjita kumara swain et al. [67] proposed an approach for generating the test data. They first created the transition graph from the state chart diagram. The test cases are generated by extracting the required information from the state chart.

2.25 REGRESSION TESTING OF OBJECT ORIENTED SYSTEM

In this section a review of various research papers related to regression testing of object oriented software is presented.

David C. Kung et al. [68] proposed an algorithm for generating order of tests of affected classes. They used an object relation graph which described all the relations existed in the object oriented program such as inheritance, aggregation, association etc.

Tarun Dhar Diwan et al. [69] proposed a technique to select test cases from regression test suite by analyzing the dynamic behavior of the application. In the proposed technique they combined the code based technique and model based technique.

Chhabi Rani Panigrahi et al.[70] proposed a regression test selection technique for object oriented programs which is based on analysis of source code of program and UML state machine model of the affected classes . They construct a dependency model of original program of source code and updated the same constructed model to reflect the changes done in the source code. The proposed model also captured control and data dependencies arising from object relation. They also constructed a forward slice by using selection criteria of the constructed graph model in intent to find the model elements affected due to program changes.

Gregg Rothermel et al. [71] proposed an algorithm to construct dependency graphs for classes and programs to determine the affected tests from exiting test suits and independent of program specification and methods.

Gregg Rothermel et al. [72] proposed a technique for selection of test case for regression testing for C++ software. In the proposed technique graph representation of software is constructed. The test cases are selected from the original test cases by using constructed graph. The selected test cases are used to execute code that has been changed for the new version of software. This technique is purely code based without any assumption for any approach that is used to specify software initially.

Alessandro Orso et al. [73] presented an RTS algorithms by consisting of two phases for Java programs which is safe, Precise and yet scales to large system. The two phases are Partitioning and Selection. The partitioning phase constructed a graph representation of programs P and P’ and analyzed the graphs to identify the parts that may be affected by changes.

Yanping Chen et al. [74] presented a specification based method for selecting test case for regression testing. The proposed approach selects two types of test cases. These types are targeted tests and safety tests. Targeted test cases exercise the important affected attribute and safety test cases are selected to reach the pre- defined coverage.

Sheng Huang et al. [75] considered the new features which are not considered yet for selecting the test cases for regression testing of an J2ee application .These features are Hybrid test case tracing and unified change identification.

Subhrakanta Panda et al. [76] proposed a method to decompose a Java program in to packages, classes, methods, and statements which are affected due to modification in the software.

On the basis of hierarchal characters of Java decomposition of program is performed. The new test cases and add some new test cases by mapping the decompositions with the existing test cases. The affected packages, classes, statements, are identified by traversing the intermediate graph. The System dependency [77] graph model is used to detect changes in the method of a program which occurs due to data dependency, control dependency and dependency caused by object relations. For verification of any statement, slicing is performed on a constructed graph.

David Binkley [78] proposed a regression testing approach based on the program slicing. Program slicing is a useful tool for working on the incremental regression testing problem.

Swapan Kumar Mondal et. al. [79] proposed an approach to minimize the regression test cases of the object oriented software based on the impacted classes. They used the optimal page replacement algorithm to minimize the test cases.

Sapna P. G. et al. [80] proposed a black box approach for generating the test cases for the regression testing. The UML and activity diagrams have been used to model the requirements and elaborated the functionality. They used the steiner tree algorithm with the objective to generate the minimal test set which are used to check functionality.

Gregg Rothermel et.al [81] proposed a regression test selection technique that is based on analysis of both the source code of the object oriented program as well as the UML state machine models of the affected classes.

2.26 TEST CASE PRIORITIZATION FOR OBJECT ORIENTED SYSTEM

In this section a review of various research papers related to the prioritizing the test cases of object oriented software is presented.

Mohammad Rava et al. [82] presented the review study of various types of technique to prioritize the test cases. They observed that all presented approach has a common combination of coverage and faults detection. The primary concern of the prioritization technique is shifted from the code analysis to history based. By reviewing the work in area of test case prioritization they also observed that the industry has adopted the artificial technique to prioritize the test cases rather than coverage based. But as the size of program exceeds a certain amount artificial technique drastically loose effectiveness.

Sun-Woo Kim et al. [83] presented a class mutation that provides a means of assessing how appropriate test cases are developed for object oriented programs. Class Mutation is a form of OO – directed selective mutation testing.

Ranjita Kumara Swain et al. [84] proposed an approach to minimize test cases for the object oriented software by using state chart. An optimization approach [85] to test data generation for the state based software testing is presented. In the proposed approach first state transition graph is derived from the state chart diagram and extracted all the required information from state chart diagram. After this the test cases are generated. The advantage of the proposed test generation technique is that it optimizes test coverage by minimizing time and cost.

Chhabi Rani Panigrahi et al. [86] prioritized the test cases by analyzing a dependency model of object oriented program. They firstly create the intermediate dependency model of program. The model is updated to reflect the change when the program is modified. The union of forward slice corresponding to each changed model is constructed for determining the affected nodes. The test cases are selected on the basis of covering the one or more affected nodes and then prioritizing on the bases of weight of the test cases.

The study of multi- objective test case prioritization technique [87] for highly configurable system address two limitation of test case prioritization technique for highly configurable system. First one is that the current prioritization technique is driven by single objective and second is that they used synthetic data to evaluate instead of industry strength case studies.

Jian Ding et al. [88] presented the comparison of two test case prioritization techniques, Adaptive random testing (ART) and dynamic random testing (DRT). They found that both techniques are extension of the random testing. ART is good for detection of failure where as DRT is good at understanding the faults. Both the technique used the different heuristics.

Rubing Haung et al. [89] presented an aggregate strength prioritization strategy for interaction test suite. The proposed technique combined the interaction coverage at different strengths whereas fixed strengths prioritization technique used the high coverage at fixed strength.

Dan Hao et al. [90] presented a unified test case prioritization approach. The presented approach includes two models. They showed that there is a spectrum of test case prioritization techniques. The spectrum is generated by the model that resides between the techniques using purely total or purely additional strategies. They proposed extensions to enable the use of probabilities that test cases can detect errors for methods and use the dynamic coverage information in place of static coverage information.

Vincenzo Martena et al. [91] proposed a technique for the inter class testing by using of data flow analysis for driven a suitable set of test specification

2.27 CODE BASED TEST CASE PRIORITIZATION TECHNIQUES

In this section various code based test case prioritization techniques is presented.

Mohammad Shahid et al. [92] presented an algorithm for test case prioritization based on code coverage. They showed that test cases that cover more methods have the higher probability to detect faults earlier.

R.Beena et al. [93] proposed coverage based test case selection and prioritization. They clustered the test cases into three groups outdated, required and surplus. Then by using these clusters test case selection algorithm (TCS) is proposed. Then the output obtained from TCS is given as an input to test case prioritization (TCP).

Alessandro Marchetto et al. [94] presented a multi objective technique that ordered the test cases to detect the maximum faults critical to business and technical. The proposed approach takes in to account the coverage of source code, application requirement and cost to execute the test cases.

A coverage based test case prioritization technique [95] used the statement, function, path, and branch and fault coverage as a criterion to prioritize the test cases. The weight is evaluated for each test case using coverage information of considered criteria. They determined and used the average weight to prioritize the test cases. The coarse grained technique [96] is used to prioritize the test suits which are based on functional coverage. The prioritization technique is focused on how much extent the test suites are dependent on each other.

Preeti.et.al [97] proposed a test case prioritization technique for object oriented software based on the source code analysis. They consider some factors and assigned them positive weights that are used to prioritize the test cases.

Ajay Kumar Jena et al.[98] proposed an approach for generation and prioritization of the test cases for the object oriented software. They used the UML sequence and interaction overview diagrams which are further converted in to the sequence interaction graph. They also consider the impact of method, activity, criticality guard of conditions and proposed a prioritization metric.

The analysis of structure of the program [99] is used to prioritize the test cases. The considered approach consists of three processes. These processes are evaluating TIM for modules, analyzing test case coverage and identifying test case priority. The proposed approach is focused on fault proneness of the module and impact of faults by analyzing the structure of program

Chabbi Rani Panigrahi et.al. [100] presented a model based test case prioritization for object oriented programs. The presented model based TCP represents the objects relations. They consider the affected elements of program as well as the elements which are indirectly tested by test case for prioritizing the test cases.

2.28 FACTORS BASED TEST CASE PRIORITIZATION TECHNIQUE

In this section various test cases prioritization using the factors has been presented

Sanjeev Patwa et al. [101] presented the factors of coding phase that effects the testing of object oriented software. These factors are programmer and tester skills, programmer and tester organization, development team size, program workload

(Stress), domain knowledge and human nature (Mistake or work omission). Analysis of factors and place of these factors according to their impact in the software are identified by using the relative weight method and ANOVA test.

A testing effort prioritization technique [102] is presented to rank components at the code level. The technique prioritized the components on the basis of five factors of the components. The considered factors are influence, average execution time, structural complexity, severity, and value. The proposed method helps tester to find the bugs in early phases.

R Karisnamoorthi et al. [103] presented a model that prioritized the system test cases. The test cases are prioritized on the basis of the six factors. These factors are the customer priority, change in requirements, implementation complexity, completeness, traceability and fault impact.

R. Kavita et .al. [104] presented an algorithm for prioritizing the test cases. They used the rate of fault detection and fault impact to prioritize the test cases. The presented algorithm determines the faults at the earlier stage of the testing process.

An algorithm is presented [105] to prioritize the system level test cases on the basis of the factors, customer priority, changes in requirement, implementation complexity, requirement traceability, and execution time and fault impact of requirement. The presented approach works at levels. At first level the requirement are prioritized and at the second level prioritization of test cases are performed.

Anup Abhinna Acharya et al. [106] presented a novel technique to prioritize the test cases. They determined business criticality value (BCV) of the functional and non functional requirements presented in the software. By using the fault model and BCV of functions the prioritization of test cases performed. They compared the proposed approach using APFD method and found that it detects the maximum faults as compared with the random test case prioritization.

Thillaikarasi Muthusamy et. al. [107] presented an algorithm to reorder the test cases to detect the maximum faults. They discussed prioritization algorithm based on four groups of weight factors. These factors are customer allotted priority; developer observed code related complexity, change in requirements, fault impact, completeness and traceability.

Soumen Nayak et al. [108] proposed a test case prioritization technique to improve the fault detection rate. They considered the four factors for prioritizing the test cases which are test case effectiveness, rate of fault detection, number of faults detected and test case ability of risk detection.

A history value based approach to prioritize [109] the test cases used the past history information to determine the present cost and fault severity for cost –cognizant test case prioritization. The outcomes of the experimented results prove it usefulness and effectiveness.

The Requirement [110] based system test case prioritization technique with equal weight for factors considered the factors Requirement change, fault impact, completeness and reusable requirement to prioritize the test cases. Each factor has assigned the weight within the range of scale 1 to 10. To calculate the weight of the factors previous testing information is used.

Monika Tayagi et al. [111] proposed a regression test case prioritization technique using three factors. The considered factors are rate of fault detection, percentage of fault detected and risk detection ability.

The system level test case prioritization technique [112] used Time, Defect, Requirement and complexity factors to prioritize the test cases. The proposed algorithm is validated by using the defect severity, Acceptable test case size and total prioritization time metrics.

Sahar Tahvili et al. [113] proposed a novel technique to prioritize the test cases. They combined the TOPSIS Decision making with principal of fuzzy. The discussed method is based on many criteria such as probability fault detection, execution time and complexity. For evolution of efficiency of test cases they used the fault failure rate as an indicator to compare the capability of fault detection with the other set of test cases.

Everton L. G. Alves et al. [114] presented the Refactoring based approach (RBA) to prioritize the test cases. The presented approach first determined the modification introduced in two version of software and collected the methods that might be impacted by change. They analyzed the impact and reorder the test cases for regression testing.

Md. Junaid Arafeen et al. [115] investigated the effectiveness the requirement based clustering based approach for prioritizing the test cases. They performed an empirical study using two Java programs having multiple versions and requirement document. The result of the study shows that the use of requirement information to prioritize the test case is very effective.

Hema sarikanth et al. [116] presented the study of prioritization of the test cases of build acceptance tests for an enterprise cloud application. Their prioritization process is based on the historical data of field failure. They found that the two or three interacting services have a tendency to be involved in the field failure.

Debasish Kundu et al. [117] generate the test cases from UML 2.0 sequence diagram and prioritize them by using the model information encapsulated in sequence diagram. Three different prioritization metrics were proposed for prioritization of the test cases. They also presented an approach for generating the test data by using rule based matrix.

2.29 TEST CASE PRIORITIZATION BASED ON VARIOUS ALGORITHMS

In this section various test cases prioritization based on various algorithms is presented.

Sangeeta Sabharwal et al. [118] proposed a technique for prioritizing the test cases scenarios by identifying the critical path clusters by using genetic algorithm. They derived the test cases scenarios form the state chart diagram and UML activity diagram. For calculating the information flow complexity associated with each node of the activity diagram and state chart diagram information flow metric is adopted.

A heuristic – based regression test case prioritization [119] technique prioritize the test cases in the base of the analysis of dependency model of the source program. The Technique construct an intermediate dependency model of a program and use this model to determine the affected nodes which are updated in the model after making modification. The union of forward slicing corresponding to each change in model is used to determine the affected nodes in the constructed model. The test cases are selected on the basis of the covering the affected nodes and further prioritized on the basis of weight assigned to the affected nodes.

Samaila Musa1 et. al. [120] presented a technique to prioritize the test cases of the object oriented software. The technique is based on analysis of dependency graph model and use the generatic algorithm to optimize the selected test cases. The test cases are ordered by computing the fitness value using the previous history of fault severity.

A model based [121] test case prioritization prioritizes the test cases on the basis of the analysis of clusters. The test cases are ordered using the degree of the preference. Unsupervised neural network and fuzzy c-means clustering algorithms are used to make the preference group. The preference degree is determined of each test case by computing mean of clustering of event using 13 attributes.

Abu Bakar Md Sultan et al. [122] presented a regression test case prioritization for object oriented systems based on the dependence graph model of affected program using genetic algorithm. ESDG (Extended System Dependency Graph) was proposed to find the statement level changes in the source code. The identified changes are stored in a file named changed and coverage information for each test case is generated from source code. Then the selected test cases are prioritized using genetic algorithm.

A meta – heuristics [123] techniques used to optimize and prioritize the test cases. The technique comprised the genetic algorithm and particle swarm algorithm. Initially the generating algorithm generates the initial population randomly and genetic operators are applied on population. The output of the genetic algorithm is given to the particle swarm optimizer as input.

Surendera Mahajan et al. [124] presented a test case prioritization technique for component based software module level testing. They developed the component based software prioritization framework with the objective to detect the more extreme bugs at earlier stage and quality enhancement by using the genetic algorithm and java decoding technique. For prioritization they proposed prioritization keys which are project size, scope of the code, information stream, bug inclination and impact of bug and faults.

Shaloni Ghai et al. [125] proposed a test case prioritization technique using hill climbing approach. They prioritized the test cases according to their functional importance. Functional importance is calculated using automated slicing.
S. Kumar Mohapatra et al. [126] used the ant colony optimization algorithm to reduce the test cases. For experimental validation the proposed approach has been applied on various programs implemented in java. The findings of the experiment show more promising results as compared to other reduction algorithm.

S. Raju et. al. [127] proposed a requirement based system level test case prioritization technique to find out the maximum error in early stage. They used the genetic algorithm to improve the quality of software. They considered the factors such as customer priority, change in requirement, implementation complexity, completeness, traceability and fault impact.

The structural testing technique [128] is generate the test cases. For generating the test cases, a genetic algorithm is applied. The generating test cases cover its def- use associations. The structural testing technique used the K Mean clustering algorithm to categorize the generated test cases in the different groups.

Ahlam Ansari et al. [129] proposed an approach for regression test case prioritization approach using ant colony optimization algorithm. The approach firstly takes the test cases which have covered the maximum faults followed by the selection of test cases covering the remaining faults.

Erum Ashraf et al. proposed [130] a value based practical swarm intelligence algorithm for prioritizing the test cases. They introduced the combination of the six factors for performing the test case prioritization. These factors are the customer priority, Requirement volatility, implementation complexity, requirement traceability, execution time and fault impact of requirement. Every factor has assigned a positive weight value in the range of 1 to 10.

Gregg Rothermel et al. [131] transformed software architectures in to intermediate representation called architectures component dependence graph (ACDG). A slicing algorithm was presented which is based on marking and unmarking the in–service and out-service edges on an ACDG, dependencies arises and occurrence of events.

2.30 TEST CASE PRIORITIZATION USING RISK FACTORS

In this section various test case prioritization technique on the based on the risk factors are discussed.

A test case selection and prioritization technique [132] using the 0-1 integer programming is presented to minimize and prioritize the test cases. The proposed approach is based on requirement priority, risk severity and statement coverage. The test cases are selected from the test suite using given time constraint. The selected test cases are prioritized using the value of requirement and risk. The 0 -1 programming is used as each decision variable and have 1 for selection and 0 for non selection.

Miso Yoon et al. [133] proposed a technique to prioritize test cases through correlation of requirement and risk. They find out relevant test cases by calculating the risk exposure value of requirement and by analyzing risk items. The basic concept of the risk based testing is to have more focus on area of software which has higher risk exposure rather than other area.

Charitha Hettiarachchi et al. [134] presented risk based test case prioritization technique. The risk related to the requirements is estimated by using the fuzzy expert system. From the result outcome it has been observed that proposed approach can detect maximum faults earlier in highly risk components compared to other techniques.

Wasiur Rahman et al. [135] proposed a model for prioritizing the test cases based on fuzzy logic. For capturing the behavior of the system, state diagram and risk information associated with the test cases is used. They classified the test cases in resettable, reusable and obsolete.

Hema Srikanth [136] et al. proposed a requirement based test prioritization technique using risk factors. They extended their earliest approach PORT 1.0 to PORT 2.0. They used two factors customer priority and fault proneness to prioritize the test cases. From the experimental outcome they observed that there is a strong correlation between CP and FP. In addition to use of two factors CP and FP they also presented a risk based system level test case prioritization.

2.31 TESTING TOOLS OF OBJECT ORIENTED SYSTEM

In this section a review of various research papers related to testing tool of object oriented software is presented.

The GenRed[137] tool is used to reduce the number of test cases and for achieving high code coverage. This tool is based on three approaches: input on demand creation, coverage based method selection, and sequence based reduction technique. This tool overcomes random testing techniques.

A frame work to test object [138] oriented programs from em formal specification to em test data generation by specifying in Z notation of object oriented program has been presented by the Ming-chi Lee. The dynamic behavior of object oriented program is represented by driven a state transition diagram (STD) from Z specification. By using of STD test data are generated. A testing algorithm modeled by em finite machine is also proposed to run again test data.

Tao Xie et al. [139] proposed a framework named Diffut for differential unit testing of object oriented programs. The proposed framework simultaneously executes the pair of corresponding methods from the two versions. The method takes the same input and framework compares the output of methods. The framework automatically generates the wrapper classes and inserts the annotations of the java modeling language.

The Framework [140] proposes a scheme of incorporating test support code as built-in test (BIT) components and also encapsulating them into framework's hot spots so that defects caused by modification and extension of framework can be easily detected through testing. A framework consists of frozen spots and hot spots. Frozen spots can be shared among applications and hot spots can be adapted or extended according to application. So whenever a framework is extended or adapted for reuse it must be tested for progressive and regressive faults. Thus by using those BIT components through testability of framework can be increased.

Jehad Al Dallal et al. [141] presented a technique to build test suite for hook methods and also introduces an automated testing tool for testing process. The presented tool has four inputs. These inputs are framework under test, formal hook description, the hook under test and select data generation.

Taweesup piwattanapong et. al [142] presented a technique for comparing the two versions of object oriented programs based on a representation. The representation can handle the features of object oriented and captures the behavior of object oriented programs. The proposed technique identified the difference and correspondence between the programs. They also proposed a tool called J Diff for implementation of the technique. The tool is used for Java programs.

Amie L. Souter et. al [143] presented the code based testing and analysis testing tool for object oriented software. This tool provides a systematic approach for testing towards behavior of object and particularly intergradations testing of class.

The CASE tool is used to support cluster [144] level testing. They also blueprint the design and implementation of CASE tool and discussed the analysis for pointer and reference.

Jitenedra S. Kushwaha et al. [145] developed and automated testing tool for object oriented software. The proposed automated testing tool includes test case generation, test case execution test data generation reporting and logging results. The proposed work mainly focused on testing design specification for object oriented software.

Anna Derezinska et al. [146] presented the C# mutation testing system that supports object – oriented mutation operators . In this paper they discussed the advances in the CREAM2 including code parsing improvement , preventing generation of invalid and partially of equivalent mutants , cooperation with distributed tester environment . They performed experiments and showed that the new version of CREAM2 system generate object oriented mutants more precisely than the previous one conducted at three levels which are unit, integration and system testing. The main components of testing tool are test order generation, test case generator for state based class testing and change impact identification for classes.

Christian Engel et al.[147] identified about integrating verification and testing techniques of object-oriented software. KEY verification system has been used to integrate both of these techniques. KEY is a system written in Java for deductive verification of object-oriented software. KEY currently integrates with two CASE tools: Boroland Together and Eclipse IDE. A whole software project can be developed with either of CASE tools and KEY verification component can be used for verification of software.

Bor Yuan Tsai et al. proposed an approach of object [148] oriented class testing which is combination of functional and structural testing. For execution of functional testing based on state based testing test cases and for data analysis MCAT (Method for Automatic class testing) tool was used.

Recardo Terra et al. [149] presented domain specific language to restrict the spectrum of dependencies that are allowed in object oriented system. They also explained a checking tool. The violations of proposed constraints are detected by this tool.

Hyunsook Do et al. [150] performed an experimental study of test case prioritization techniques for java programs tested under JUnit testing framework. The results show that test case prioritization techniques can significantly improve the rate of fault detection of JUnit test suites.

The technique for selective regression testing and associated tool for object oriented software is [151] based on the concept of control call graph. The technique used static analysis of code of the program. The developed tool combined with impact analysis identifies impacted call paths that needed to be retested, select the test cases from an existing test suite and generation of new test cases if required.

2.32 CONCLUSION

From the critical review of the above literature, it has been observed that the various researchers presented their work for performing the effective testing of object oriented software. Almost in the every concern related to the testing of object oriented software, various researchers proposed their techniques whether it is complete testing of a software, regression testing , prioritization of test cases and automated tools for testing the software. They considered the important factors that affect the testing of object oriented software. A pointed overview is shown below

(1) The various researchers used slicing of program, program dependency graph, control dependency graph, data dependency graph, directed graph for performing testing.
(2) The researchers also used model, data flow, state, fault, specification, coupling for performing effective testing of the software.
(3) They also considered and proposed various metrics like DCH, MRC, dynamic metrics, design metrics, object oriented metrics, classification of metric and a basic metric of object oriented design.

(4) Impact of coupling and cohesion, security concern, software complexity also considered.
(5) They also used genetic algorithm, ant colony , hill climbing algorithm etc. to prioritize the test cases.
(6) Researchers also presented some issues related to the testing of the object oriented software. They classified the problem related to the testing of object oriented software
(7) Researchers presented the relation between the cyclomatic complexity and degree of object orientation.
(8) Researcher presented security issues related to architecture of object oriented software and a security model for assessment.

In the previous work there are some critical issues related to the testing of the object oriented software that are not discussed yet. There should be design metric on which issues related to design may be tested. There is no any framework and technique that reduces the cost and time for testing the software. In object oriented software there are some critical factors which play critical role in developing the software. If these factors are not used in proper way they might affect the working of the software. These factors are exception handling, multithreading, use of pure virtual function, virtual function etc. There should be software metric on which prioritization of test cases is performed with the intent to the find the errors early. Some of these issues have been discussed in this work which is presented in the next chapters.

Chapter III

UNIT AND INTEGRATION TEST CASE PRIORITIZATION TECHNIQUES: PROPOSED WORK

3.1 INTRODUCTION

In this chapter, test case prioritization techniques to prioritize the test cases at Unit testing, Integration testing of object oriented software is presented. The four techniques are proposed to prioritize the test cases. The proposed techniques use some factors to prioritize the test cases. These proposed techniques are

- A Multi - Factored Cost and Code Coverage Based Test Case Prioritization Technique for Object Oriented Software.
- A Structural Analysis based Test Case Prioritization Technique for Object Oriented Software.
- Test Case Prioritization Technique for Object Oriented Software Using Method Complexity.
- A Coupling Analysis based Test Case Prioritization Technique for Object Oriented Software.

All the proposed techniques are explained and validated by applying on some case studies in the subsequent sections.

3.2 A MULTI - FACTORED COST AND CODE COVERAGE BASED TEST CASE PRIORITIZATION TECHNIQUE FOR OBJECT ORIENTED SOFTWARE (MFCCTCPTOOS)

The presented approach prioritizes the test cases on the basis of the cost and the coverage of the code covered by the test case. For accurately finding out the cost of the test case, some factors are considered as shown in the Table 3.1. The proposed approach works at two levels. At the first level all the considered factors existed in the source code are identified. After identification and counting the factors all independent paths of the source code are resoluted then the value of the cost of each path is determined on the basis of the coverage of the identified factors. Test cases are selected corresponding to independent paths. The cost of the test case can be calculated by using Formula 3.1.

The code coverage of test case is determined by counting lines of code executed by the test case. At the second level pairs of cost and code value of each test case are created. In this way by using the value of the cost and code coverage the test cases are prioritized. The following scenario is used for prioritization of the test cases

(1) Highest code coverage and cost will have highest priority

(2) Second priority is given to test case that has highest cost value on the basis of covered factors

(3) Third priority is given to test case that has highest code coverage.

(4) Test cases with the equal code coverage and cost be ordered

The overview of the proposed approach is shown in Figure 3.1

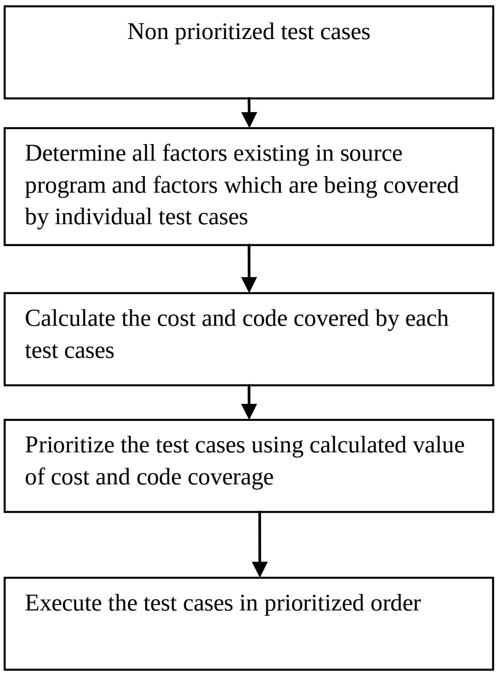

Non prioritized test cases
Determine all factors existing in source program and factors which are being covered by individual test cases
Calculate the cost and code covered by each test cases
Prioritize the test cases using calculated value of cost and code coverage
Execute the test cases in prioritized order

Figure 3.1: Overview of the Proposed Approach (MFCCTCPTOOS)

$$\text{Cost}(T_i) = SF(T_i) / TF \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (3.1)$$

Where SF is the sum of the factors covered by the ith test case, TF is the sum of the all existing factors in source code.

3.2.1 Considered Factors For Prioritizing Test Cases

As shown in Table 3.1, the factors are considered by the structural analysis of the program. The considered factors may affect the testing process in term of consumption of memory, execution time and the possibility of introducing the errors in program.

Table 3.1: Proposed Factors (MFCCTCPTOOS)

S. No.	**Proposed Factor**
1	Operators
2	Variables
3	External System Call
4	Predicate Statement
5	Assignment Statement
6	Use of Libraries/ Packages
7	Virtual Function/ Functions
8	Exception Handling
9	Other Factors

The algorithm of the proposed approach is given in Figure 3.2.

3.2.2 Result and Analysis

For the experimental validation and evaluation, the proposed approach has been applied on the two programs. The programs are implemented in the C++ language. For the experimental analysis intentionally faults are introduced in the programs. The program one (see Appendix A) has 170 lines of code, program [152] two has 361 lines of code.

Table 3.2 shows the various factors covered by the test cases, Table 3.3 shows the line of code covered by the test cases, Table 3.4 shows the calculated cost of all test cases that are used to test the software, Table 3.5 shows the various pairs of cost and code covered by the test cases.

```
Let T be is the list of non prioritized test cases and T' be the list of the prioritized test cases.

While ( T not empty)

Begin

Step 1.  Identify and Count all the considered factors that are used in the source code.

Step 2.  Determine the factors and line of code being covered by the test cases.

Step 3.  Calculate the cost by applying the formula on test cases.

                 $Cost (T_i) = SF(T_i) / TF$

Where SF is the sum of factors covered by the test case and TF is the sum of the factors in the
source  code

End

Step 4. Determine all possible pairs of the code coverage value and cost value of each test case.

                        Pair  = (Code Coverage, Cost)

Step 5. Prioritize the test cases in the following scenarios

(1) Highest the value of cost and code covered by the test case have highest priority

(2) Second priority is given to test case that has highest cost value.

(3) Third priority is given to test case that has highest code coverage.

(4) Test cases with the equal value of the code coverage and cost be prioritized in the random
order.

Create T' the list of prioritize test cases.
```

Figure 3.2: Algorithm of the Proposed Approach (MFCCTCPTOOS)

Table 3.2: Factors Covered by Test Cases

Factors	**TC1**	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**	**TC7**	**TC8**
Operators	4	4	0	0	1	7	0	0
Variable	3	3	1	4	2	3	1	4
Native method	0	0	0	0	0	0	0	0
Control statement	0	1	0	0	0	2	0	0
Assignment	3	2	0	0	1	2	0	0
SF	10	10	1	4	4	12	1	4

Table 3.3: Line of Code Covered by Test Cases

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
Line of Code	36	42	31	36	34	48	31	36

Table 3.4: Calculated Cost of Test Cases

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
Factor Coverage (SF)	10	10	1	4	4	12	1	4
Total Factors(TF)	35	35	35	35	35	35	35	35
Cost	0.2857	0.2857	0.0285	0.1142	0.1142	0.3428	0.0285	0.1142

Table 3.5: Pairs of the Cost and Code Coverage by Test Cases

S. No	Test Case	Pairs
1	TC1	(36, 0.2857)
2	TC2	(42, 0.2857)
3	TC3	(31, 0.0285)
4	TC4	(36, 0.1142)
5	TC5	(34, 0.1142)
6	TC6	(48, 0.3428)
7	TC7	(31, 0.0285)
8	TC8	(36, 0.1142)

The prioritized order of test cases as determined by the proposed approach is TC6, TC2, TC1, TC4, TC8, TC5, TC3, and TC7.

Faults Detected by Test Cases in Non Prioritized Order

The faults are identified in the non prioritized order as shown in Table 3.6.

Table 3.6: Faults Detected by Test Cases in Non Prioritizing Order

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
F1	*	*	*	*	*	*	*	*
F2				*				*
F3								*
F4					*			
F5						*		
F6						*		
F7				*				
F8	*							
F9		*						
F10		*						

Faults Detected by Test Cases in Prioritized Order

The Table 3.7 shows the faults detected by the test cases when they executed in prioritized order

Table 3.7: Faults Detected by Test Cases in Prioritize Order

	TC6	TC2	TC1	TC4	TC8	TC5	TC7	TC3
F1	*	*	*	*	*	*	*	*
F2				*	*			
F3					*			
F4						*		
F5	*							
F6	*							
F7				*				
F8			*					
F9		*						
F10		*						

For simplicity of the approach the faults are detected for only one program. After this the comparison of two approaches in term of APFD is shown in Figure 3.3 and Figure 3.4.

3.2.3 Comparison of APFD Graphs of Prioritized and Non Prioritized Order of Test Cases for Two Programs

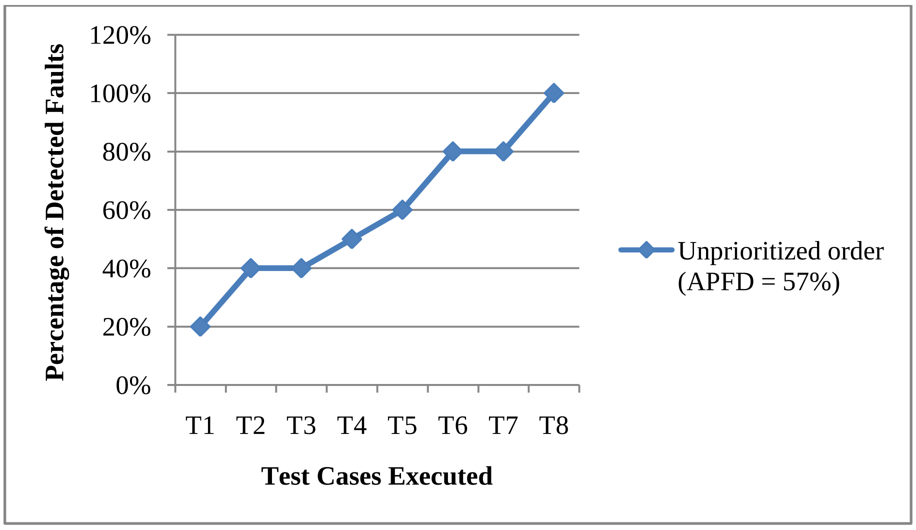


Figure 3.3: APFD Graph of Non Prioritized Order of Test Cases

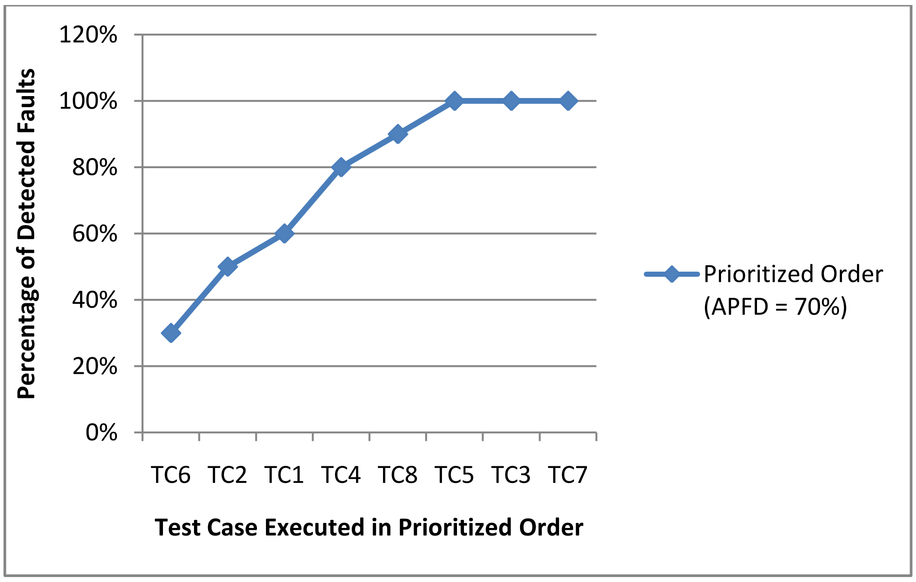


Figure 3.4: APFD Graph of Prioritized Order of Test Cases

The same approach was applied on a Program 2 of Income Tax Calculator [176] implemented in C++ programming Language. The considered program has 361 lines of code. For experimental validation of the approach 24 faults have been added intentionally and detected by 19 test cases. The APFD of non prioritized order and prioritized order of the test cases is shown in Figure 3.5 and Figure 3.6.

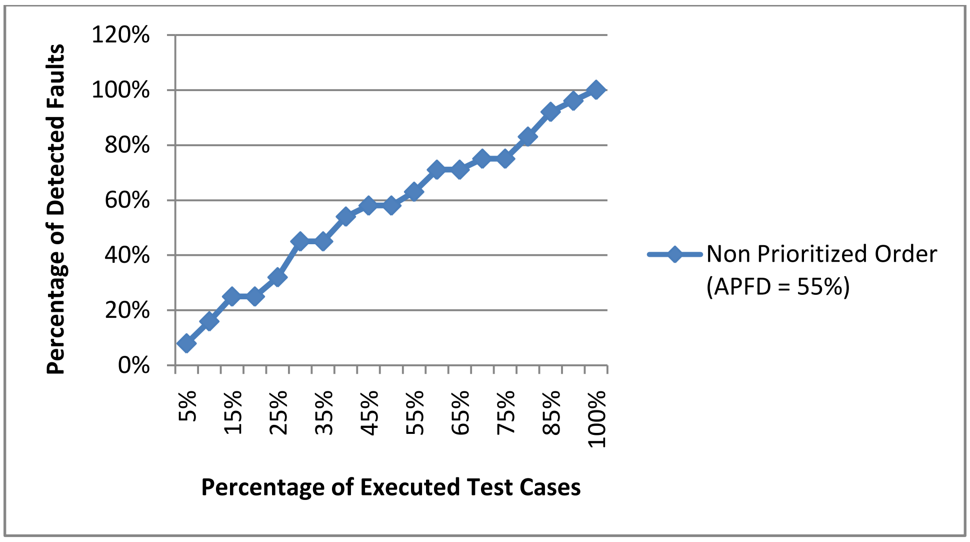


Figure 3.5: APFD Graph of Non Prioritized Order of Test Cases

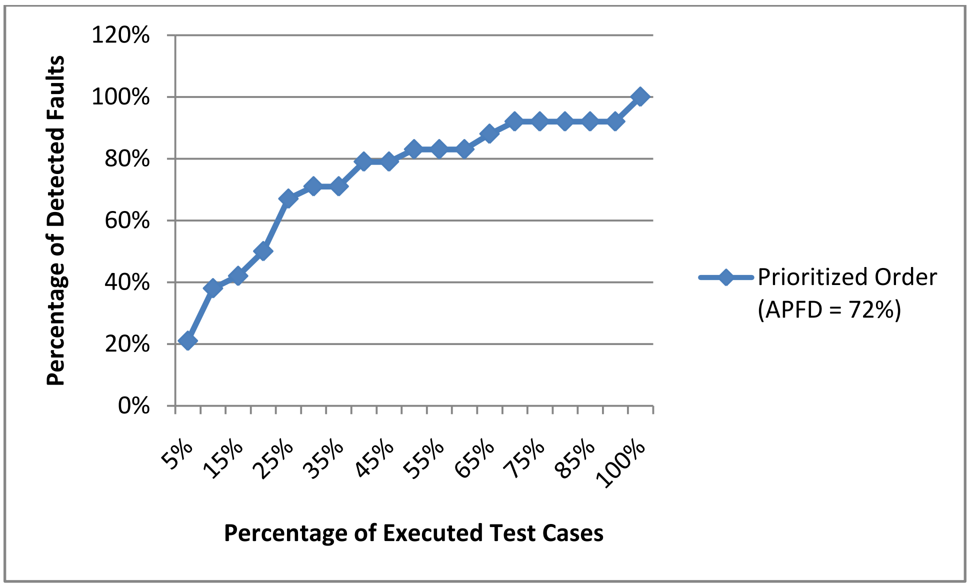


Figure 3.6: APFD Graph of Prioritized Order of Test Cases

3.2.4 Effectiveness of the Proposed Approach

The effectiveness of the proposed approach is measured through APFD metric and its value is shown in Table 3.8. The APFD value of prioritized order of test cases obtained by applying the proposed approach is better than non prioritized order of test cases. Therefore it can be observed from the Table 3.8 prioritized test cases has higher fault exposing rate than the non prioritizing test cases.

Table 3.8: Compared Result of Test Cases for Prioritized and Non Prioritized Order

Case Study	**Non Prioritized Test Cases (APFD)**	**Prioritized Test Cases (APFD)**
Program 1	57%	70%
Program 2	55%	72%

3.3 A STRUCTURAL ANALYSIS BASED TEST CASE PRIORITIZATION TECHNIQUE FOR OBJECT ORIENTED SOFTWARE (SATCPTOOS)

The proposed approach works at three levels. At the first level intermediate representation of the program objected oriented control flow graph (OOCFG) is created by analysing the structure of the program. At the second level by analysing the OOCFG graph all the independent paths of a program are determined and there by test cases are selected corresponding to every independent path. Finally at the third level the test cases are prioritized on the basis of coverage of factors. The overview of the proposed approach is shown in Figure 3.7

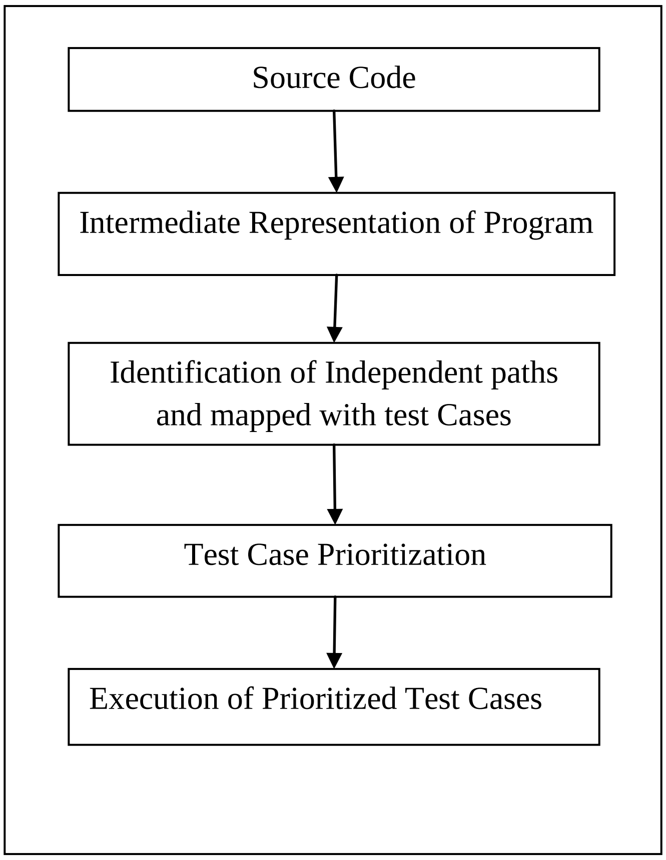

Source Code
Intermediate Representation of Program
Identification of Independent paths and mapped with test Cases
Test Case Prioritization
Execution of Prioritized Test Cases

Figure 3.7: Overview of Proposed Approach (SATCPTOOS)

All the considered factors have been assigned a weight on the basis of possibility of faults introduced by the factors. To verify and assign the significant weight of factors a survey (see Appendix C) has been performed. The conducted survey focuses on factors, which affect the testing of the software. The participants involved in survey are the software developer, tester, tech lead etc. having average experience of 8 years in software industries. To examine the view of participants the survey questionnaire was submitted among several software developers and testers in various software industries.

Based on the criticality of the factors a weight is assigned to proposed factors. The assigned weight shows the capability of introducing the errors in the program. The weight metric of the proposed factors are as shown below in the Table 3.9.

Table 3.9: Proposed Factors and Assigned Weight (SATCPTOOS)

S. No	**Factor**	**Weight**
1	Class/Interface	.05
2	Type Casting	.15
3	Exception handling	.3
4	Method overriding	.2
5	Native method	.1
6	Nested class	.05
7	Conditional Statements	.05
8	Number of method	.1

The discussion of the proposed factors is given below

- **Class:** Class is a basic unit for the test case design. The intended use of a class implies different test requirements. Testing a class instance can verify a class in isolation. However when these verified classes are used to create objects in an application system must be tested as whole before it can be considered verified

- **Type Casting:** Type casting is a way of changing one type of data into the other type of data. During the development of software sometimes, it is essential to convert the type of data to fulfil the customer expectation or requirement for implementations of the software. In past practice it has been observed that many big projects failed or crashed due to some mathematical errors. Sometimes software is very hard bound to their numerical value. They don't deal any type of minor mistake in the value. At the time of development the programmers are not aware about the consequence of the error but later it may become the reason of causing many errors thereby increasing the cost and development time of software. So in the development if the developers are using the type casting they should properly test whether they have given the proper formatted value or not.

- **Exception handling:** Exceptions handling is procedure of responding to the occurrences of the exceptions. Basically the exceptions are the error that arise either at the run time or compile time. For instance in Java the exception that comes at compile time is called checked exception whereas that exception that comes at run time is called unchecked exception. During the computation of programme, exceptional conditions often change the normal flow of the program execution. If the properly exceptions are not handled at the right time they may corrupt the data.

- **Native Method:** Native methods are the methods which are implemented in other language and used in the current language used for developing the software. e.g. the functions implemented in C/C++ are used in the Java language. For executing the native method the libraries of native method are required. So, there may be higher chance of occurrences of the errors due to use of native method. Because native methods are implemented in other language so it is very hard to detect the error. So if there is any use of native method then it must be tested properly.

- **Method overriding:** Method overriding is used to provide a specific implementation to a method in subclass that is already provided by one of its base class. The method in the super class and the override method in the base class have same name, same parameters, and same return time. At run time the object will determine which version of the method is executed. So, there is need to be proper calling of the method. If the proper version of method is not executed as per requirements then it will give wrong results. So, there is need to design the test cases to test all possibility of correctly invoking of all different versions.

- **Method:** There may be a chain of methods to implement the user requirements. A tester needs to understand the sequence of the methods invocation and also design the test accordingly to the sequence of methods.

- **Conditional Statements:** Each condition in a decision may have possible outcomes. The coverage of condition does not mean that the decision has been covered. It requires adequate test cases such that each condition in a decision takes on all possible outcomes at least once. So it is essential to take all the possible combinations of the conditions that are used in the application system.

- **Nested Classes:** Nested class is the special feature of the object oriented language. Nested class is used to resolve the issue of the naming and for purpose of containment. There is no requirement of special testing if nested class is viewed as naming issue but in lieu of purpose of containment a multilevel testing strategy is required.

3.3.1 Representation of the Program in Intermediate Form

In this section program is represented in the intermediate representation. For representation of program some symbolic notation are presented which are shown in the Figure 3.8. The intermediate representation shows execution flow of the program. Since the program structures of object oriented program are different from the conventional program so here some representations are presented which represent the features of the Object Oriented Programming System (OOPS) i.e. class, interface, method, method overriding, nested class, exception handling etc.

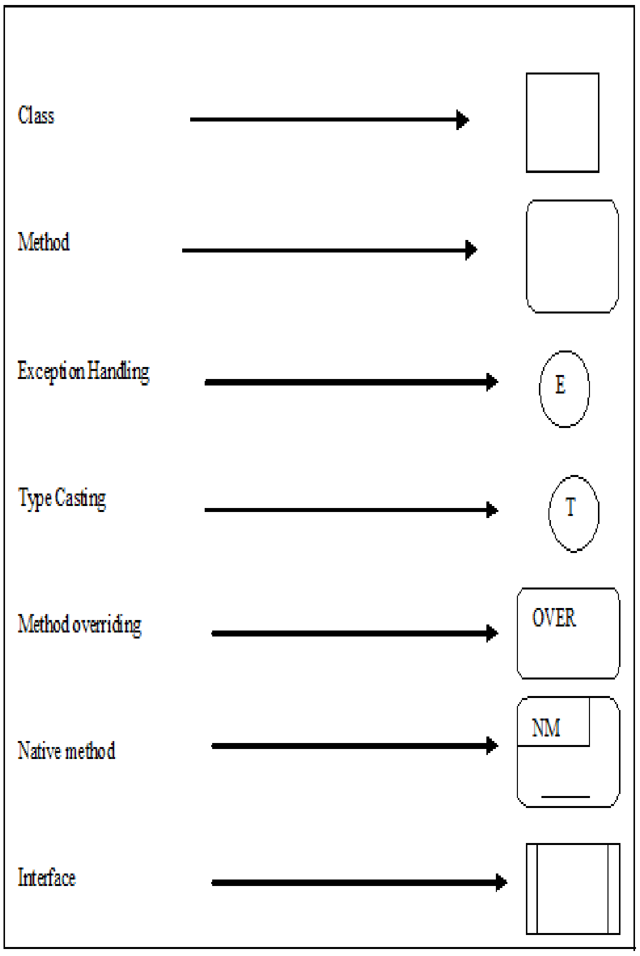


Figure 3.8: Representation of Various Features

3.3.2 Identification of Independent Paths

By using the representations showing in the Figure 3.9 the OOCFG of program is created which are further analysed to identify all the independent paths. After determining all the independent paths are mapped to test cases.

3.3.3 Test Case Prioritization

Mapped test cases are prioritized on the basis of the proposed 8 factors. The test cases are prioritized on the basis of the coverage of the factors. Test case with the highest coverage value has the highest priority of execution as these factors show the critically of the test case based on coverage of factors. Thus a test case with highest criticality will have the higher probability of error to be found out.

By using the Table 3.10 these test cases are prioritized using the Formula 3.2

$$TCPW = \sum_{i=1}^{n} fvalue_i * fweight_i \text{ ----------------------------(3.2)}$$

where fvalue is the values of factors covered by test cases, fweight is the weight assigned to the factor which shows the criticality of the factor , TCPW is the calculated weight of the test cases. On the basis of TCPW the test case are prioritized. More the complexity of the test cases more the probability of the error to be detected by test cases.

3.3.4 Result and Analysis

For evaluation and analysis of proposed approach it has been verified and analyzed by applying on a case study of (Appendix A) software. The considered case study performs the various functionalities like to calculate the gross salary, saving, deduction, taxable income, and tax to be paid by the employee. To determine the efficacy of the proposed approach some faults have been added in the software intentionally .The OOCFG of considered case study are shown in Figure 3.9.

After analyzing the Figure 3.9 independent paths are determined. To test the considered software each and every independent path need to be tested. So test cases should be selected or designed for each independent path. All the independent paths and Ids of test case are shown below in Table 3.10.

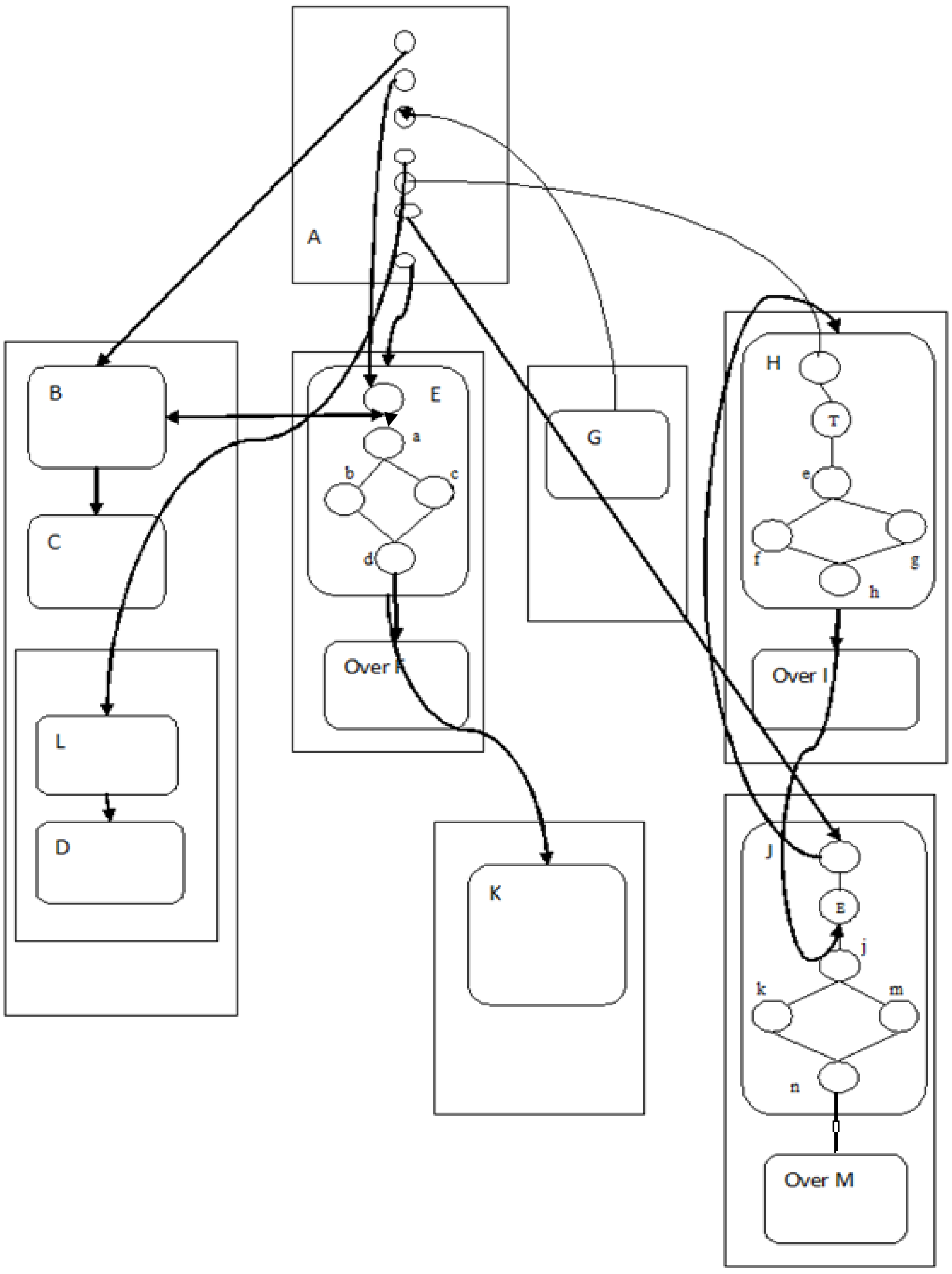


Figure 3.9: OOCFG of Considered Case Study

Table 3.10: Independent Path and Test Cases Corresponding to the Independent Paths

S.No.	**Independent path**	**Test case ID**
1	A,B,C	TC1
2	A,L,D	TC2
3	A,E,B,Ea,Eb,Ed, F	TC3
4	A, E, B, Ea,Ec,Ed, F	TC4
5	A,E,B,Ea,Eb,Ed, k	TC5
6	A, E, B, Ea,Ec,Ed, F	TC6
7	A, G	TC7
8	A, H,He,Hf,Hh,I	TC8
9	A, H,He,Hg,Hh,I	TC9
10	A,j, H,,J,Jj,Jk,Jm, M	TC10
11	A,j, H,J,Jj,Jk,Jn, M	TC11

After determining the independent paths and mapping the test cases corresponding to all paths, now test cases are prioritized. The Table 3.11 shows the factors covered by the test cases.

Table 3.11: Factors Covered by the Test Cases

S. No	Factors to be covered	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10	TC11	Weight Of Factors
1	**No. of Class**	2	2	3	3	4	0	2	2	2	3	3	.05
2	**No. of Nested Class**	0	1	0	0	0	0	0	0	0	0	0	.05
3	**No. of method**	2	2	3	3	4	0	1	1	1	2	2	.1
4	**No. of override method**	0	0	1	1	0	0	0	1	1	1	1	.2
5	**Exception Handling**	0	0	0	0	0	0	0	0	0	1	1	.3
6	**Type Casting**	1	0	1	1	1	0	0	1	1	1	1	.15
7	**No. of Native Method**	0	0	0	0	0	0	0	0	0	0	0	.1
8	**Conditional Statement**	0	0	1	1	1	0	0	1	1	1	1	.05

The test case are prioritized on the basis of TCPW obtain by applying the Formula 3.2. Highest value of TCPW of the test cases highest the priority of the test case to be executed. The Table 3.12 shows the TCPW of the all selected test cases.

Table 3.12: Calculated Value of TCPW.

S. No	Factors to be covered	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10	TC11	Weight Of Factors
1	**No. of Class**	.1	.1	.15	.15	.2	.2	.1	.1	.1	.15	.15	.05
2	**No. of Nested Class**	0	.05	0	0	0	0	0	0	0	0	0	.05
3	**No. of method**	.2	.2	.3	.3	.4	.4	.1	.1	.1	.2	.2	.1
4	**No. of override method**	0	0	.2	.2	.2	.2	0	.2	.2	.2	.2	.2
5	**Exception Handling**	0	0	0	0	0	0	0	0	0	.3	.3	.3
6	**Type Casting**	.15	0	.15	.15	.15	0	0	.15	.15	.15	.15	.15
7	**No. of Native Method**	0	0	0	0	0	0	0	0	0	0	0	.1
8	**Conditional Statement**	0	0	.05	.05	.05	0	0	.05	.05	.05	.05	.05
	TCPW	.45	.35	.85	.85	1.0	1.0	.2	.6	.6	1.05	1.05	1

By using the calculated TCPW of each test case form Table 3.12 the prioritized order of the test cases are TC10, TC11, TC5, TC6, TC3, TC4, TC8, TC9, TC1, TC2, TC7.

Fault Detection in Non Prioritized Order

The Table 3.13 shows the detected faults when the test cases are executed in the non prioritized order.

Table 3.13: Faults Detected in Non Prioritized Order

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10	TC11
F1	*		*	*	*	*					
F2	*										
F3		*									
F4			*		*						
F5			*								
F6				*		*					
F7					*	*					
F8							*				
F9								*	*	*	*
F10								*		*	
F11								*	*		
F12										*	
F13											*
F14										*	*

Calculated APFD for non prioritized order of Test cases: 52%

Fault Detection in Prioritized Order:

The Table 3.14 shows the detection of faults when the test cases are executing in the prioritizing order which obtain after applied the approach

Table 3.14: Faults Detected in Prioritized Order

	TC 10	**TC 11**	**TC5**	**TC6**	**TC3**	**TC4**	**TC8**	**TC 9**	**TC1**	**TC2**	**TC7**
F1			*	*	*	*			*		
F2									*		
F3										*	
F4			*		*						
F5					*						
F6				*		*					
F7			*	*							
F8											*
F9	*	*					*	*			
F10	*						*				
F11							*	*			
F12	*										
F13		*									
F14	*	*									

Calculated APFD for Prioritized order of Test cases: 64.5 %

The APFD graph shown in Figure 3.10 and Figure 3.11 shows that the APFD value obtained from the proposed approach is better than the non prioritized approach. The result shows the efficacy of the proposed approach.

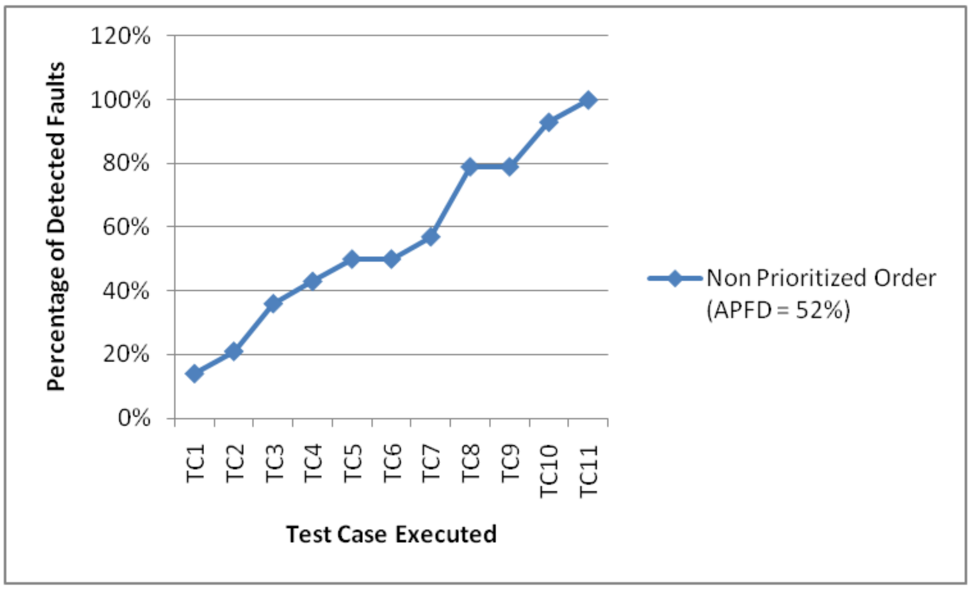


Figure 3.10 APFD Graph for Non Prioritized Approach

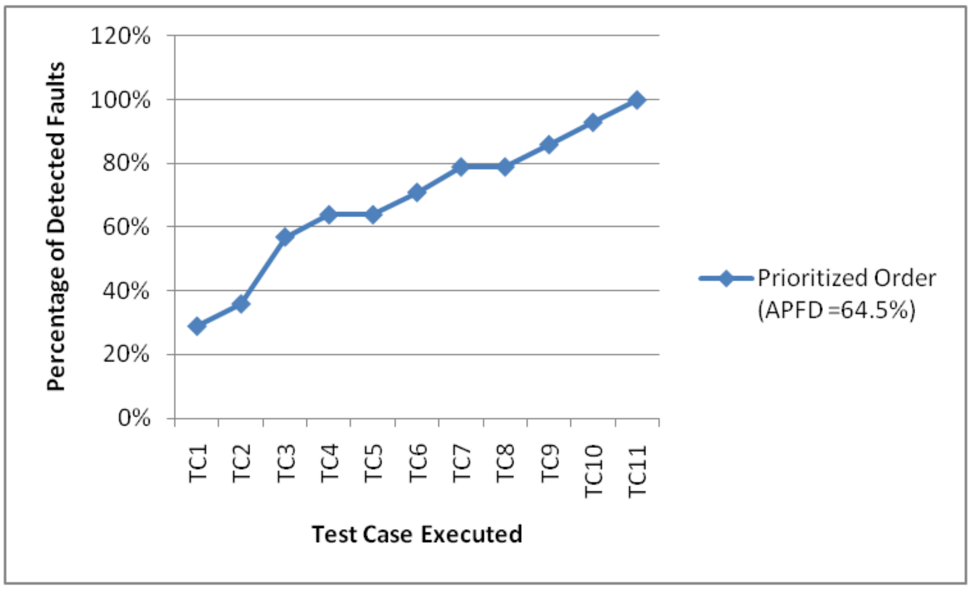


Figure 3.11: APFD Graph for Proposed Approach

The same approach was applied on another football player information system [153] that implemented in the C++. The APFD graph of comparison of the proposed and non prioritized approach is shown in Figure 3.12 and in Table 3.15.

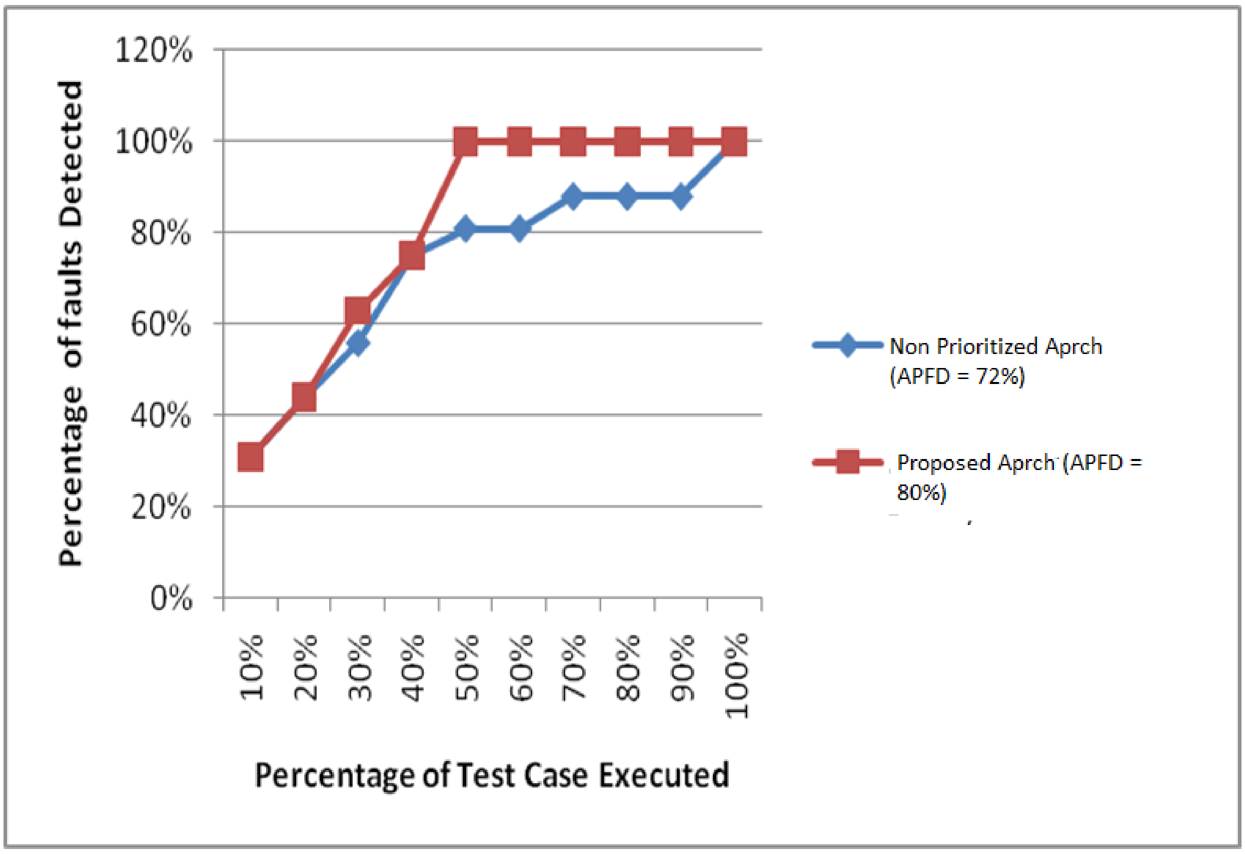


Figure 3.12: Comparison between the Proposed Approach and Non Prioritized Approach

Table 3.15: Case Study 2 Results (APFD)

Projects	**Strategy**	**APFD results**
Case Study2	Non Prioritized	72%
	Proposed	80%

3.4 TEST CASE PRIORITIZATION TECHNIQUE FOR OBJECT ORIENTED SOFTWARE USING METHOD COMPLEXITY (TCPTOOSUMC)

In the presented approach firstly source code is represented in the intermediate form called the method call graph (MCG) followed by the determination of the complexity of the each method used in the call graph. The complexity of a method is calculated by using volume and difficulty of a method, which are further determined by the factors identified by the structural analysis of the source code. The factors which are used to determine the method complexity are given in Table 3.16.

Table 3.16: Considered Factors and Assigned Weight (TCPTOOSUMC)

S.No	**Factor Name**	**Weight**
1	Degree of Method(DM)	0.6
2	No. of Input Variable(IV)	0.3
3	Decision statement (DS)	0.4
4	Type Casting(TC)	0.6
5	Numerical computations(NC)	0.4
6	Number of loop(LS)	0.5
7	Number of variable reused (VR)	0.2
8	Copying of objects (CO)	0.3
9	Object/Data reads from database/File(RW)	0.6
10	Exception handling (EH)	0.7
11	Virtual function (VF)	0.9
12	Dynamic memory allocation and deallocation (MA)	0.8
13	Reference counting (RC)	0.2
14	Proxy Objects (PO)	0.3
15	Type binded inherited Function (TIF)	0.8
16	Copy constructor having pointer type variable (CPV)	0.4
17	Non virtual destructor (NVD)	0.2
18	Return object by reference (RO)	0.2

Every considered factor has been assigned a factor weight which indicates the difficulty to test the factor and posses the higher probability of the errors.

For determination of the weight of considered factors a survey was performed in various industries (See Appendix B). The survey was performed among Developers, Senior Developer. Technology Lead, Associate Architect Group Leader and Project Manager with an average experience of seven years. From the survey approximate 80 responses were received from participants and same data was compiled for determination of the assigned weight. The overview of the proposed approach is shown below in Figure 3.13.

For process of prioritization of test cases, value of Volume and difficulty of a method can be used. The determination of the value of the volume, difficulty and complexity of a method can be given as below:

Volume of a method (VM) can be calculated by Formula 3.3.

$$VM(m_i) = FM/TF \text{ ---------------------------- } (3.3)$$

Where FM is the number of considered factors in i_{th} method and the TF is the total count of considered factors in the whole software i.e in whole method existed in the software.

Difficulty of a method (DM) can be calculated by the Formula 3.4

$$DM(m_j) = \sum_{i=1}^{n} F_i * W_i \text{ ------------------------------- } (3.4)$$

Where F_i is the number of determined i_{th} factors in an j_{th} method and W_i is the weight assigned to the i_{th} factors.

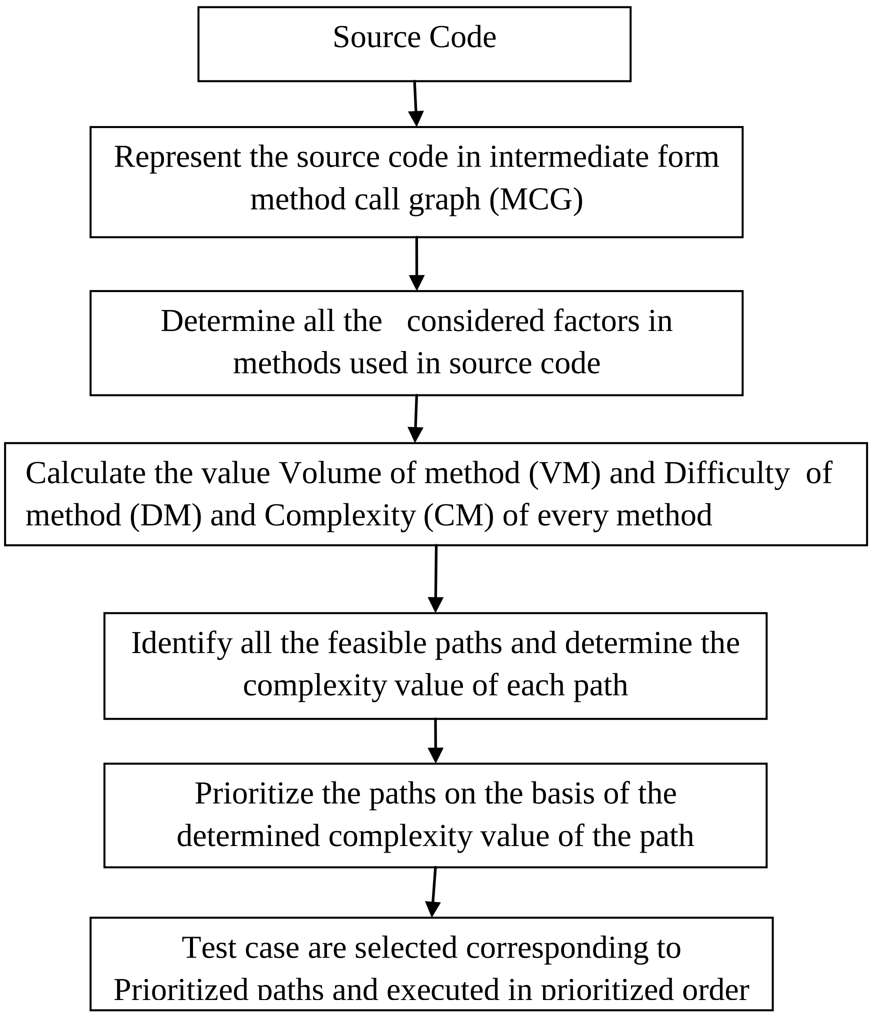

Source Code
Represent the source code in intermediate form method call graph (MCG)
Determine all the considered factors in methods used in source code
Calculate the value Volume of method (VM) and Difficulty of method (DM) and Complexity (CM) of every method
Identify all the feasible paths and determine the complexity value of each path
Prioritize the paths on the basis of the determined complexity value of the path
Test case are selected corresponding to Prioritized paths and executed in prioritized order

Figure 3.13: Overview of the Proposed Approach (TCPTOOSUMC)

Thus complexity (CM) of each method can be calculated by the Formula 3.5

$$CM(m_i) = VM * DM \text{ --------------------------(3.5)}$$

Where VM is the volume of the i_{th} method and DM is the estimated difficulty of i_{th} method.

After calculating the value of CM for all methods, all the feasible independent paths from method call graph are identified. The path prioritization value (PPV) is determined for each path which is sum of the calculated method complexity (CM) of methods that are used in the path. The PPV is calculated by the Formula 3.6.

$$PPV = \sum_{i=1}^{n} CM_i \text{-------------------------------------} (3.6)$$

More the PPV value of the path, more is the complexity of the path and in turn the higher chances of error. So paths are prioritized on the basis of the PPV. After path prioritization, test cases are selected corresponding to each path and executed in the order of the paths. If any path has more than one test case then these are prioritized on the basis of considered factors covered by the individual test case.

The algorithm of the proposed approach is shown in Figure 3.14. The presented algorithm takes the source code as an input and converts them into method call graph (MCG) by using Create_MCG function. The volume and difficulty of each method is calculated by identifying the considered factors in each method and using the Formula 3.3 and Formula 3.4.

The function Compute_complexity determines the method complexity by using the Volume of a method and difficulty of a method. The MCG is used to identify all the feasible paths in the software and determine the methods covered in each path. The output of Compute_complexity is used to calculate the path prioritization value which is further used to prioritize the test cases.

```
Let S be a source code, T be the set of non prioritized test cases and T' be the prioritized test cases.
1.  Create_MCG( S)
Find out all the methods used in the source code and create the method call graph (MCG) of source
code
 2 . while (method)
 Begin
        Determine the Volume of each method using  the formula 3.3
 End
 3.  While (method)
Begin
  Determined the difficulty of each method by using the formula 3.4
 End
4  While (method)
Begin
    Compute_complexity( VM, DM)
    Begin
        Find out the value of Complexity of each
        Method (CM) by using the formula 3.5
    End
End
5. All the feasible independent method call paths are identified.
6. Determined the PPV of the each path using the formula 3.6
7. Paths are prioritized on the basis of the determined value of PPV
8. Test cases are selected corresponding to each path and T, be the set of prioritize test cases.
```

Figure 3.14: Algorithm of Proposed Approach (TCPTOOSUMC)

3.4.1 Result and Analysis

For experimental verification and analysis, the presented approach has been applied on a billing management system [154] implemented in the C++ programming language. The considered software performs various functions like place order, create product, modified product, delete product etc. For experimental verification, intentionally some errors are introduced in the software and introduced errors were discovered by applying the proposed approach. The finding of the case study is given below

The Figure 3.15 shows the method call graph of the considered case study. In this graph, all the methods that are used are connected by using the direction arrows which shows the sequence of the calling of the methods. By analyzing the sequence of the calling methods all the feasible independent paths and methods covered in each path are determined.

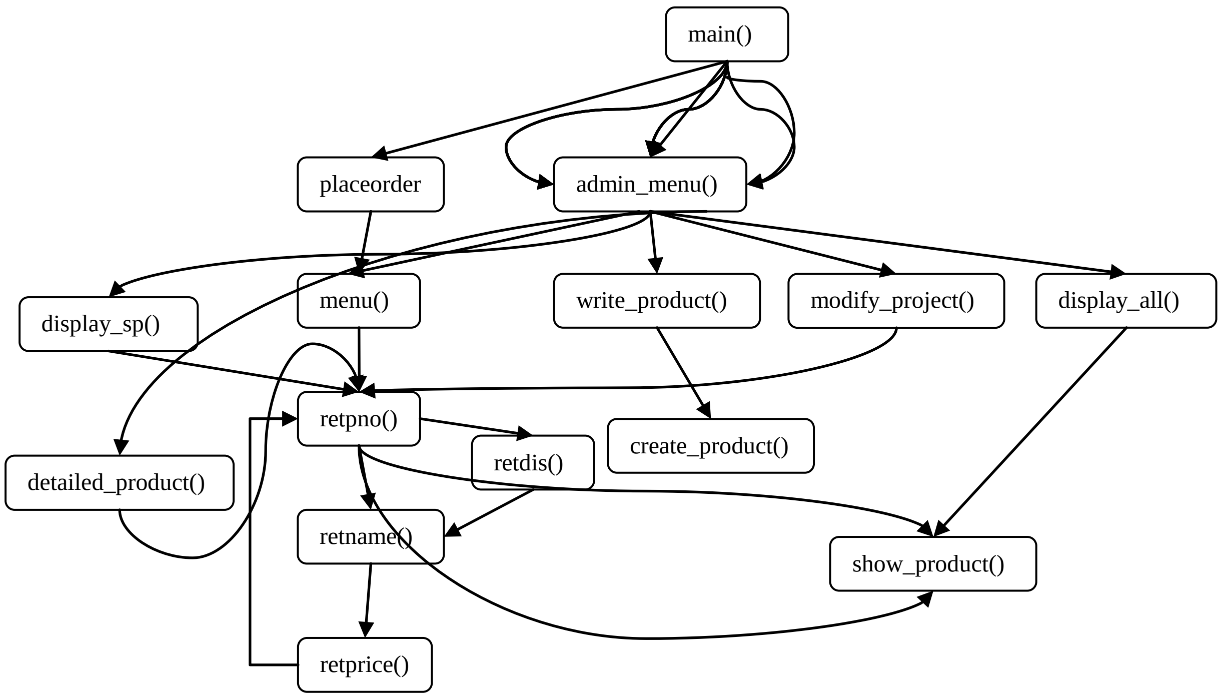


Figure 3.15: Method Call Graph (MCG) of Case Study

The Table 3.18 shows the methods used in the software and the count of considered factors identified in each method. The value of volume, difficulty and complexity of each method after computation is also given in the Table 3.17.

Table 3.17: Determined Value of VM, DM and CM

S. No.	Method Name	Factor determined	VM	DM	CM
1	Place_ order	IV=7,LS=3 ,NC=1, RW=2 NC=3,VR=1,DM=2	19/47=.40	(7*.3) +(3*.5)+(1*.4) +(2*.6)+(3*.5) +(1*.2) +(2*.6) = 8.1	3.24
2	Menu	RW =1 ,DS = 1, LS =01,DM=3	6/47=.12	3.3	.39
3	admin_menu	IV=2,CS=1,DM =12	15/47=.31	9.2	2.8
4	write _product	RW=1,DM=2	3/47=.06	1.8	.10
5	create_ product	IV =4 ,DM=1	5/47=0.1	1.4	.14
6	modify_ product	IV=3, RW=2,CS=2,LS =1,VR=1,NC=1,DM=2	12/47=.25	5.3	1.32
7	display_all	RW=1, LS=1,DM=2	4/47=.08	2.3	.18
8	show_ product	IV = 4,DM=3	7/47=0.14	3.0	.42
9	delete_product	IV=1, RW=2,LS=1,CS=1, DM=2	7/47=.14	3.6	.50
10	display_sp	RW=1,IV=2,VR =1,CS=2,DM=2	8/47=.17	3.4	.57
11	retpno	IV =1,DM=08	9/47=.19	5.1	.96
12	retname	IV=1,DM=03	4/47=.08	2.0	.16
13	retprice	IV=1,DM=02	3/47=.06	1.5	0.09
14	retdis	IV=1,DM=02	3/47=.06	1.5	0.09

Table 3.18 shows all the feasible and independent paths that are determined after analyzing the method call graph and the estimated path prioritization value of each identified path.

Table 3.18: PPV of All Feasible Independent Paths

S.No.	Path ID	Path	Estimated PPV
1	Path1	main(),palce_oreder(),menu(),retpno(),retname(),retprice() ,retdis	3.24+.43+1.12+. 18+0.10+0.10= 4.93
2	Path2	main(), admin_menu(),write_product(),create_product()	3.04
3	Path3	main(),admin_menu(),display_all(),show_product()	3.4
4	Path4	main(), admin_menu,modify_product(),retpno(), show_product	5.5
5	Path5	main(),admin_menu,display_sp,retpno(), show_product()	4.75
6	Path6	main(),admin_menu,delete_product(), retpno()	4.26
7	Path7	main(),admin_menu,menu()retpno(),retname(),retprice()	4.4

The Table 3.19 shows estimated path prioritization value of each considered path obtained from MCG and the test cases that execute the identified independent paths.

Now the test cases are prioritized on the basis of estimated cost of paths that are executed by the test cases. The prioritized order of the test suit is T3, T4, T9, T10, T5, T6, T8, T7, T2, T1. Now the proposed approach is being compared with random approach and method coverage [92] based approach. For this purpose faults are detected by executing the test cases.

Table 3.19: Paths Covered by Test Cases

S.No.	**Path ID**	**Test cases**	**Estimated Path Prioritization Value (PPV)**
1	Path4	T3,T4	5.5
2	Path1	T9,T10	4.93
3	Path5	T5,T6	4.75
4	Path7	T8	4.4
5	Path6	T7	4.26
6	Path3	T2	3.4
7	Path2	T1	3.04

The Table 3.20 shows the faults detected by test cases when these test cases are executed in non prioritized order.

Table 3.20: Fault Detected by Test Cases in non prioritized Order

Testcase	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**F 10**	**F 11**	**F 12**	**F 13**	**F 14**	**F 15**	**F 16**
T1	*			*												
T2		*			*											
T3	*	*					*	*	*							
T4							*			*						
T5		*				*										
T6						*										
T7											*	*				
T8			*										*			
T9			*								*		*	*	*	*
T10													*	*	*	*

To show the efficacy of the approach a metric called average percentage faults detection (APFD) has been used.

APFD value of non prioritized order of test cases is 54%

The Table 3.21 shows the faults detected by the test cases when these test cases are executed in prioritized order that has been obtained after applying the proposed approach.

Table 3.21: Faults Detected by Test Cases in Prioritized Order

Test case	**F 1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**F 10**	**F 11**	**F 12**	**F 13**	**F 14**	**F 15**	**F 16**
T3	*	*					*	*	*							
T4							*			*						
T9			*								*		*	*	*	*
T10													*	*	*	*
T5		*				*										
T6						*										
T8			*										*			
T7											*	*				
T2		*			*											
T1	*			*												

The APFD value of prioritized order of test case by applying the proposed approach is 69%

Table 3.22 shows the faults detected by the prioritized test cases obtained by applying the method coverage based approach.

Table 3.22: Faults Detected by Ordered Test Cases Obtained from Method Coverage Based Approach

Test case	F1	F2	F3	F4	F5	F6	F7	F8	F9	F 10	F 11	F 12	F 13	F 14	F 15	F 16
T9			*								*		*	*	*	*
T10													*	*	*	*
T8			*										*			
T3	*	*					*	*	*							
T4							*			*						
T5		*				*										
T6						*										
T1	*			*												
T2		*			*											
T7											*	*				

The APFD value of prioritized order of test case by applying the method coverage based approach is 65%

The Figure 3.16, Figure 3.17 and Figure 3.18 shows the APFD graph of random approach, method coverage based approach and proposed approach showing the efficacy of the proposed approach.

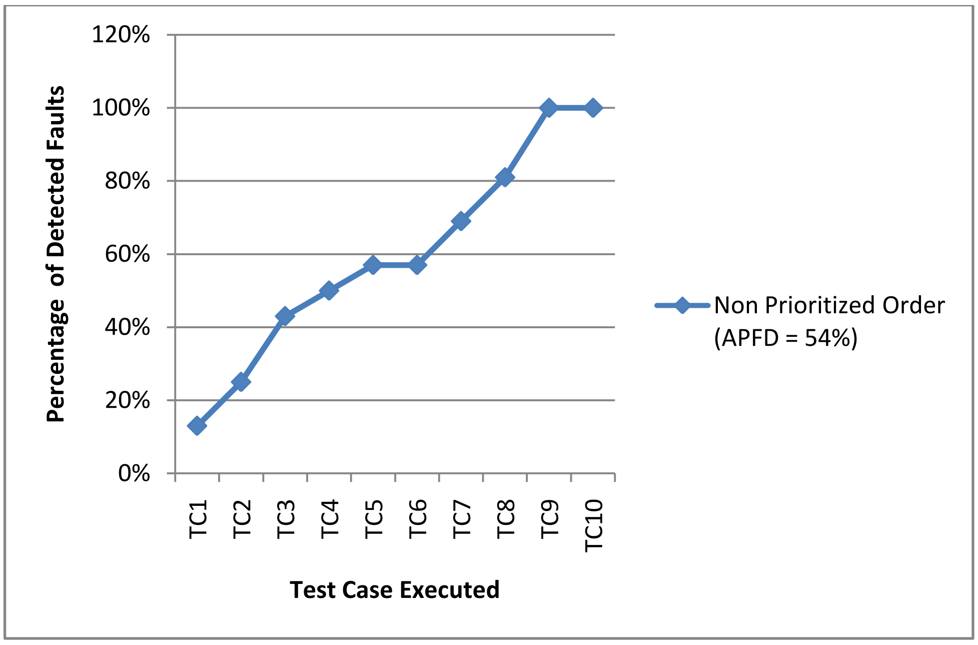


Figure 3.16: APFD Graph for non Prioritized Approach

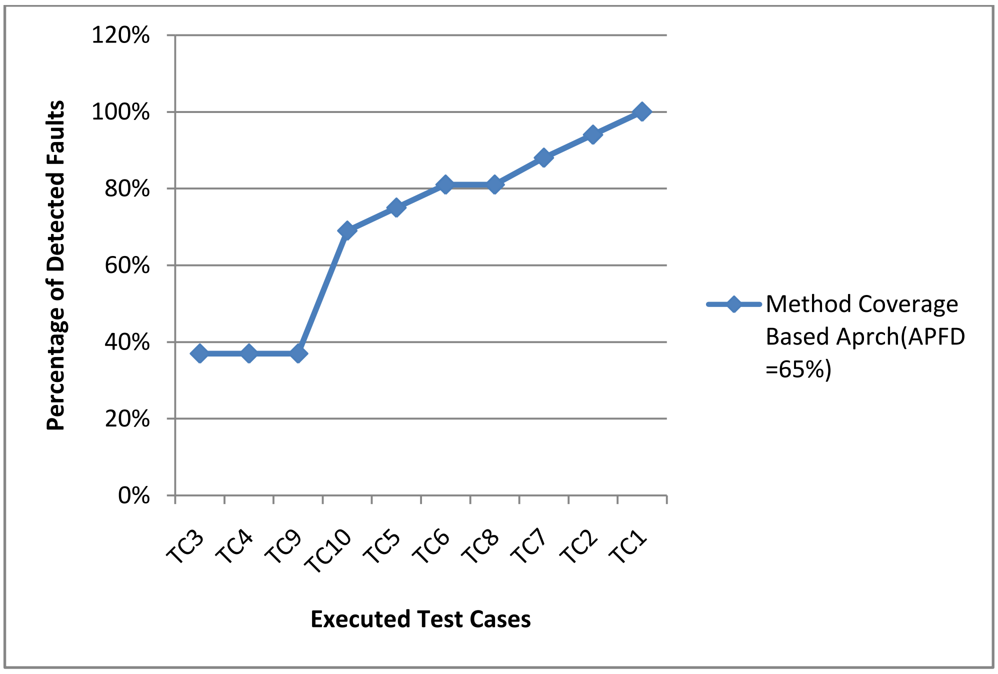


Figure 3.17: APFD Graph for Method Coverage based Approach

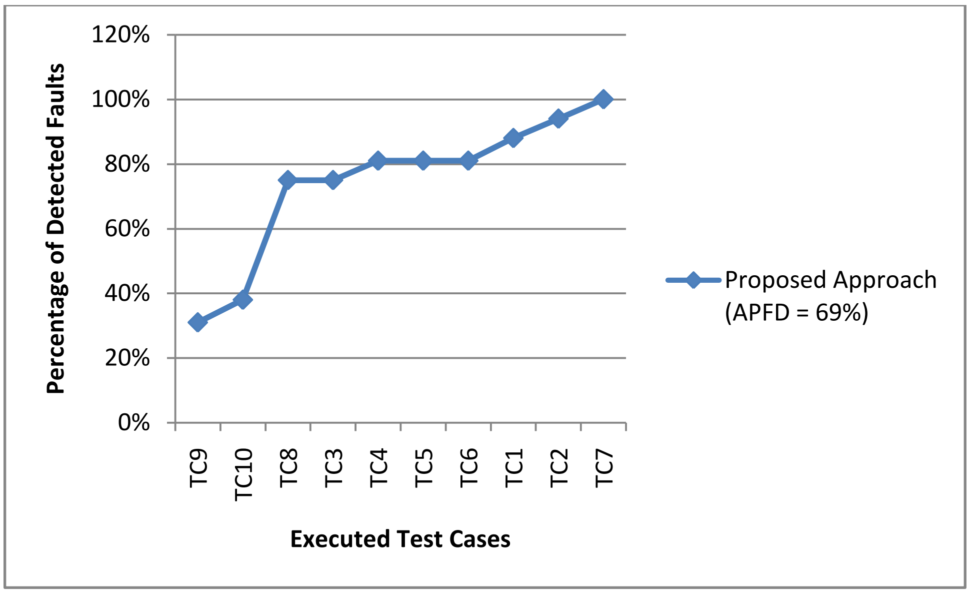


Figure 3.18: APFD Graph for Proposed Approach

The same approach was applied on another software room reservation [155] which performed all the operations related to reserve a room in hotel. The considered software has total 1936 line of code and 74 test cases are executed to detect the 56 faults, inserted intentionally. The APFD graph of the non prioritized approach, method coverage based approach and proposed approach is shown in Figure 3.19.

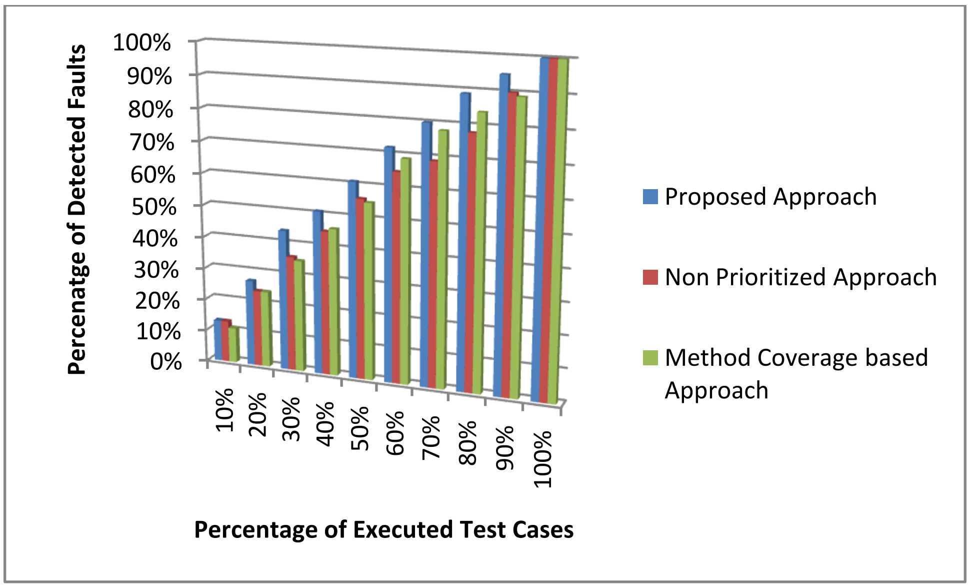


Figure 3.19: APFD Graph For Non Prioritized, Proposed And Method Coverage Based Approach For Hotel Room Reservation Software

3.5 A COUPLING – ANALYSIS BASED TEST CASE PRIORITIZATION TECHNIQUE FOR OBJECT ORIENTED SOFTWARE (CATCPTOOS)

The proposed approach works at three phases. In the first phase interaction coupling of all existed individual class is determined. In the second phase the inheritance coupling and component coupling existed between the classes existed in software is determined. In the third phase all the possible combination of the classes existed in the software are determined. The coupling value of determined combination is calculated by using the determined value of interaction coupling of individual class and inheritance coupling and component coupling existed between the classes. The calculated coupling value of each combination helps to determine the coupling value of each combination of classes. The coupling value shows, how complex and how implicit the information has to be. More the coupling value of combination of classes means the classes are tightly interconnected to each other; therefore these are very hard to test and maintain. The coupling value of each combination is used to prioritize the combination and individual classes are prioritized on the bases of interaction coupling. The prioritized order of the combination and individual classes are mapped with the test cases that executed the corresponding combination and classes. Three types of coupling exist in object oriented software [14]. These are interaction coupling, component coupling and inheritance coupling. Each coupling type has been assigned a positive weight (see in Table 3.23) which shows the worseness of the particular coupling. The couplings [14] are further sub divided under each category as shown in the Table 3.24.

Table 3.23: Value of Weight Assigned to Coupling Type

S. No.	**Coupling Name**	**Assigned weight**
1	Inheritance Coupling	3
2	Component Coupling	2
3	Interaction Coupling	1

Table 3.24: Weight Assigned to Dimensions of Coupling under Each Category

S.No.	**Dimension of coupling**	**Degree of coupling**	**Weight**
1	Interaction coupling	Content coupling	.6
		Common Coupling	.5
		External Coupling	.4
		Control coupling	.3
		Stamp Coupling	.2
		Data Coupling	.1
2	Component coupling	Hidden Coupling	.3
		Scattered Coupling	.2
		Specified Coupling	.1
3	Inheritance Coupling	Modification Coupling	.3
		Refinement Coupling	.2
		Extension coupling	.1

3.5.1 Determination of Interaction Coupling Existed In Individual Classes

In object oriented programming system, methods are coupled by interaction in terms of invocation of each other. By analysis of the individual classes the interaction coupling is determined. The interaction coupling (Cintr) value of classes is determined by using the Formula 3.8 and Table 3.25.

The degrees of interaction coupling are Content coupling, Common coupling, External coupling, Control Coupling, Stamp Coupling, and Data Coupling. Each type of coupling has been assigned a positive weight which shows the worseness of the coupling types.

Cintr(C) = (.6*Cn+.5*Co+.4*Ex+.3*Con+.2*St+.1*Dt) --------------------------(3.8)

Where Cn is the number of content coupling, Co is the value of common coupling, Ex is value of external coupling, Con is the value of control coupling, St is value of stamp coupling and Dt is value of data coupling.

3.5.2 Determination of Component and Inheritance Coupling Between the Classes

In this level coupling between the possible combinations of the classes are determined. The coupling of the combination is identified by analyzing the component and inheritance coupling existed in the combination of the classes. The value of combination is calculated by using the Formula 3.9 and Formula 3.10 as shown below

Ccomp = (.3*HC + 2*SC +.1*SPC) ------------------------------------(3.9)

Cinhr = (.3*MC + .2*RC + .1*EC) ------------------------------------ (3.10)

Where Ccomp is calculated value of component coupling , HC is the hidden coupling, SC is the Scattered coupling, SPC is Specified coupling, Cinhr is the calculated value of inheritance coupling , MC is modified coupling, RC is refined coupling and EC is extension coupling.

3.5.3 Prioritization of the Combinations of Classes and Individual Class

In the third phase test cases are prioritized. For prioritizing the test cases the combinations of the classes are prioritized by using the coupling value of combinations of classes. The Coupling value of each combination (Vcomb) is calculated by using the Formula 3.11 as given below

$$Vcomb = 3*Cinhr + 2*Ccomp + 1*CIntr \text{ ----------------------------------- } (3.11)$$

The individual classes are prioritized on the basis of the calculated value of interaction coupling.

3.5.4 Mapping of All Possible Combination of Classes and Individual Classes with the Test Cases

After prioritizing the combination of classes and the individual class a prioritized ordered of the combination of classes and individual classes is obtained. Every combination of class and individual class is covered by the one or more test case so the ordered combinations of classes and individual class are mapped with the test cases which are executed covering the classes.

3.5.5 Prioritization of the Test Cases

Test cases are prioritized on the basis of coupling value of combinations of classes. Test cases of the highest prioritized combinations are executed first. It may be possible that combinations have more than one test case. In such types of circumstances the prioritization the test cases can be done using any of earlier test case prioritization method [156, 157].

3.5.6 Algorithm.

An algorithm has been designed for the above process as shown in Figure 3.20.

```
Step 1. Let T be the set of unprioritized test cases and T' be the set of prioritized test cases.

Step 2 Let C be the set of classes that are used in the software

Step 3 Determine all the possible combination of classes that are used in software.

Step 4 Determine the interaction coupling of individual class using formula given below

    $Cintr(C) = (.6*Cn +.5*Co+.4*Ex+.3*Con+.2*St+.1*Dt)$

Step 5Determine the component coupling and inheritance coupling for all the possible combination of the classes by using the following formulae

        $Ccomp = (.3*HC + 2*SC +.1*SPC)$

        $Cinhr = (.3*MC + .2*RC + .1*EC)$

Step 6 Determine the Coupling value of each combination and individual class by using the formula

$Vcomb = 3*Cinhr + 2*Ccomp + 1* Cintr$

Where is Cintr is determined by the formula 1, Ccomp is determined by formula 2 and Cinhr is calculated by the formula 3

Step 7 Prioritize the combinations on the basis of determined value of VComb

Step8 Map the test cases with the prioritized combinations of classes and individual class.

Step 9 T' is the set of prioritized test cases

Step10 Execute the test cases in prioritized order.
```

Figure: 3.20: Algorithms for the Proposed Approach (CATCPTOOS)

3.5.7 Example: To explain the presented approach it was applied on an example as shown below in Figure 3.21.

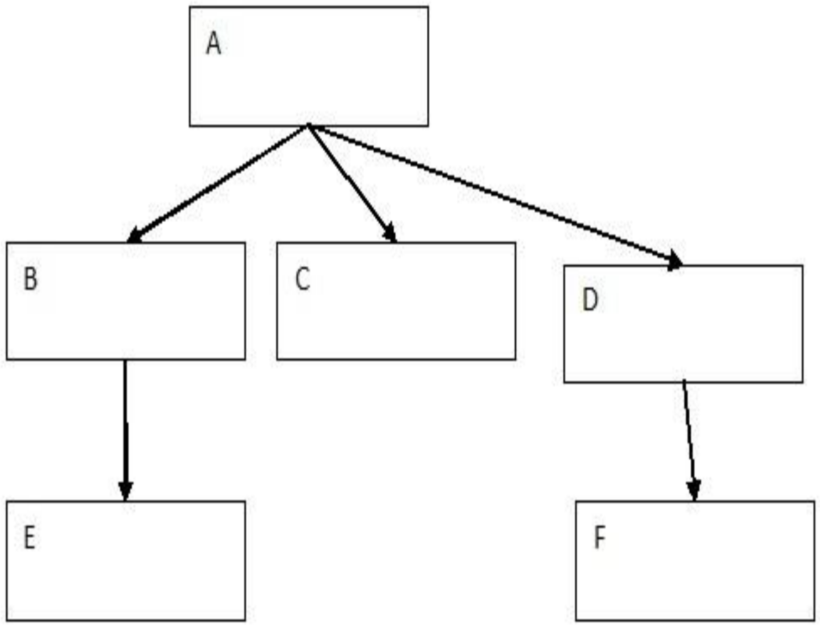


Figure 3.21: Consider Hierarchy of Classes

As above shown in the Figure 3.21 there are five class as which are interrelated to each other

Let C = {A,B,C,D,E,F}

Now make all the possible combinations of the class shown in the Figure 3.21

Sup(C)= {A},{B},{C},{D},{E},{F},{A,B},{A,C},{A,D},{B,E},{D,F}{A,B,E} {A,D,F}

At first level calculate the coupling of the individual class using the Table 3.24 and Formula 3.8. The Table 3.25 shown below shows the interaction coupling value existed in the class

Now the component coupling and inheritance coupling existed in all possible combination are identified and remove all combinations which do not have the component and content coupling. The Table 3.27 shows all the types of coupling Vcomb value of the combinations of classes.

Table 3.25: Determined Value of Interaction Coupling in Individual Classes

S.No.	**Class Name**	**Determined Interaction coupling in classes**	**Calculated Cint value of individual class**
1	A	Cn = 1, dt =3	(.6*1+.1*3) = .9
2	B	Dt = 2	(.1*2)= .2
3	C	Cn=1 ,st =4 dt =3	(.6*1 + .2*4 + .1*3)= 1.7
4	D	Ex =1	(.4*1) = .4
5	E	Dt =2	(.2 *1) = .2
6	F	Dt =3	(.3 *1) = .3

Table 3.26: Determined Value of Component and Inheritance Coupling of All Combination of Classes

S.NO.	**Combinations**	**Component**	**Inheritance**	**Ccomp value**	**Cinhr**
1	ABE	HC = 1	EC = 2	(.3*1) = .3	(.1*2)= .2
2	ADF	0	MC =1 EC =2	0	(.3*1 +.1*2)=.5
3	AB	SC=1	0	(.2 *1) = .2	0
4	AC	HC =1	EC =1	(.3 *1) = .3	(.1*1) = .1
5	AD	0	EC =1	0	(.1*1) =.1

The Table 3.27 shows the determination the Vcomb value of combination of classes and individual class

Table 3.27: Determined Value of VComb of Combination of Classes and Individual Classes

S.No	**Prioritized order of sub sets of classes**	**Value Calculation**	**Vcomb**
1	ABE	3*.2 + 2 *.3 + .9 +.2 + .2	2.5
2	ADF	3*.5 + .9 + .4 + .3	3.1
3	AB	2*.2 + .9 + .2	1.5
4	AD	3 *1 + .9+ .4	1.6
5	AC	3*1 + .2 * .3 + .9 + 1.7	3.5
6	A	1*.9	.9
7	B	1*.2	.2
8	C	1*1.7	1.7
9	D	1*.4	.4
10	E	1*.2	.2
11	F	1*.3	.3

Now using the above table prioritizing order of the combinations of the class and individual class is {A,C} {A,D,F},{A,B,E},{C},{A,D},{A,B},{A},{D},{F},{E},{B}

Now performing the mapping between the test cases and the combinations of classes and individual class as shown below in the Table 3.28.

Table 3.28: Prioritized Combination of Classes, Individual Class and Covered Test Cases

S.No	**Prioritized order of sub sets of classes**	**Sub sets covered by test cases**
1	AC	T7
2	ADF	T11
3	ABE	T,12, T13,
4	C	T3
5	AD	T10,T8
6	AB	T9,
7	A	T1,
8	D	T4
9	F	T6
10	B	T2
11	E	T5

The prioritized order of the test cases is T7, T11, T12, T13, T3, T10, T8, T9, T1, T4, T6, T2, T5,

3.5.8 Result and Analysis.

To analyze the efficacy of the proposed approach, it was applied on a software [158] dispensary management system. The software was implemented in C++ language. The considered case study has drags, cost, save data, load data, change data and start class. The finding of the result after applying the approach has been given below.

Table 3.29: Determined Interaction Coupling of Individual Class

S.No	Class name	Determined coupling dimension
1	Drags	0
2	Cost	Cn =1
3	Saving data	0
4	Load data	0
5	Change data	0

Table 3.30: Determined Component and Inheritance Coupling Interaction Coupling of Individual Class

S.No	**combinations**	**Cinhr**	**Ccomp**	**Cintr**	**Mapped test case**
1	Cost, Drags	EC =1	HC =1	Cn =1	T5
2	Load data, Saving data	EC = 1	HC = 1	0	T4
3	Saving data, Cost	EC=1	HC =1	Cn=1	T3
4	Changedata, Saving data, Cost	EC=3	HC=3	Cn =1	T2
5	Drags	0	0	0	T1

Table 3.31: Calculated Value of Vcomb

S.No	**Combinations**	**Vcomb**
1	Changedata, Saving data, Cost	2.1
2	Load data, Saving data	.9
3	Saving data, Cost	1.5
4	Cost, Drags	1.5
5	Drags	0

The prioritized order of test cases is T2, T3, T5, T4, and T1.

The Table 3.32 shows the faults detected when the test cases are executed in the non prioritized order

Table 3.32: Faults Detected by the Test Cases in Non Prioritized Order

	TC1	TC2	TC3	TC4	TC5
F1	*	*	*	*	*
F2		*	*	*	*
F3		*	*	*	*
F4			*	*	*
F5				*	
F6			*		
F7					*
F8					*

The Table 3.33 shows the faults detected when the test cases are executed in the prioritized order.

Table 3.33: Faults Detected by the Test Cases in Prioritized Order

	TC2	TC3	TC5	TC4	TC1
F1	*	*	*	*	*
F2	*	*	*	*	
F3	*	*	*	*	
F4		*	*	*	
F5				*	
F6		*			
F7			*		
F8			*		

The Figure 3.22 and 3.23 shows the APFD graph of faults detected by the test cases in non prioritized order and prioritized order.

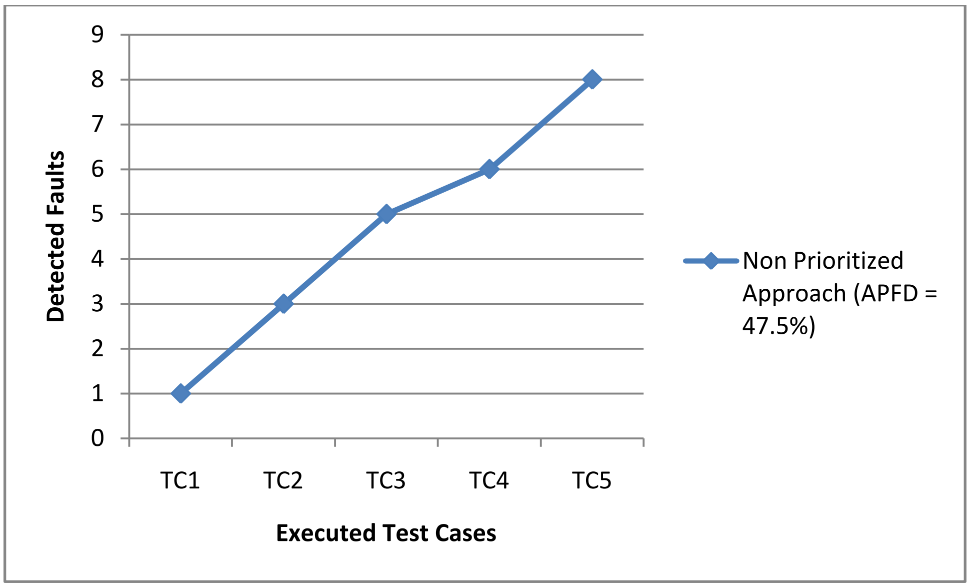


Figure 3.22: APFD Graph for Non Prioritized Approach

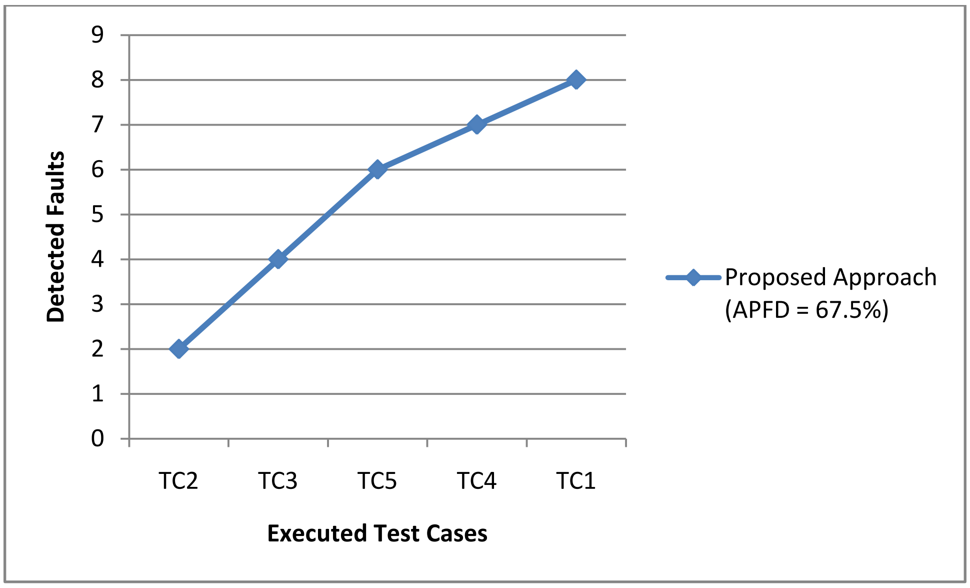


Figure 3.23: APFD Graph for Proposed Approach

The proposed approach is also applied on the software of advance payroll management [159]. The considered software was implemented in Java and performs various operations like addition of employee, edit, deletion, change settings, generate slips. To check the validity of the proposed approach 16 faults are added in the software and detected by executing the 18 test cases. The proposed approach was also compared with the other existing approach [98]. The Figure 3.24 shows the comparison of APFD graph of execution of ordered test cases obtained by applying the non prioritized approach, Ajay Kumar Jena[98] and proposed approach.

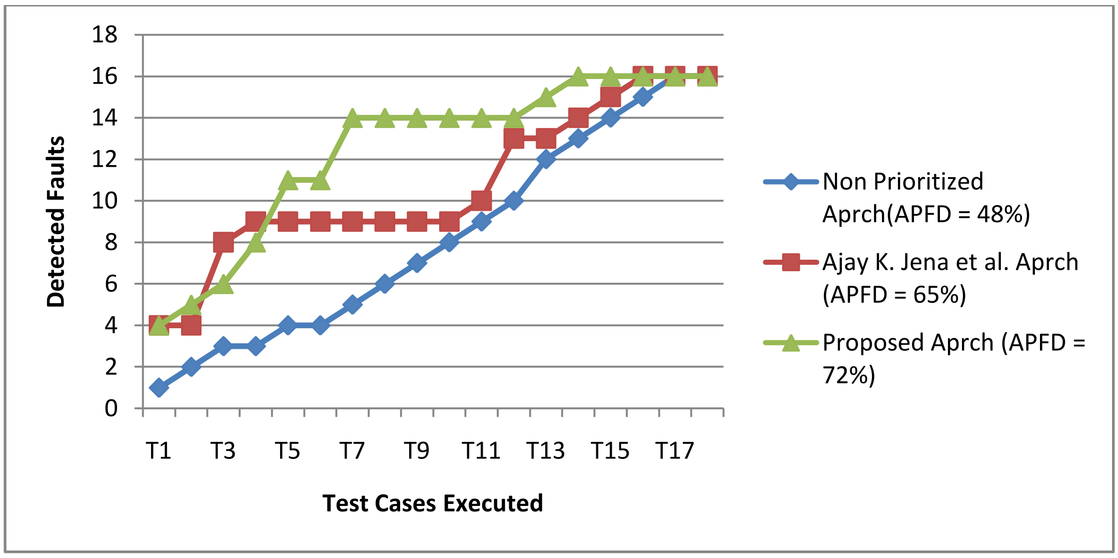


Figure 3.24: Comparison of APFD Graph for Non Prioritized, Proposed and Ajay k. Jena Approach

3.6 CONCLUSION

In this chapter four techniques are presented to prioritize the test cases of the software implemented using the OOT. Every technique considers some factors which help to detect the maximum faults as early stages as possible. First technique prioritizes the test cases on the basis of the cost and code and considers some factors to determine the cost. In the second technique OOCFG for the object oriented software is proposed, which uses some factors representation and used their weight to prioritize the test cases. The third technique prioritizes the test cases on the basis of the complexity of the methods. Some factors are considered to calculate the method complexity. In the fourth technique coupling existed in the object oriented software is used to prioritize the test cases. For experimental verification and validation all the approaches have been applied on the various software. The finding of the analysis shows the effectiveness of the proposed approaches.

Chapter IV

SYSTEM TEST CASE PRIORITIZATION: PROPOSED WORK

4.1 INTRODUCTION

In system testing the software is required to be tested in the real conditions which are very challenging. So large numbers of test cases are generated and executed. It is very expensive and time consuming process to execute all test cases. In this chapter, a technique to prioritize the system test cases for object oriented software and a cost reduction framework for the same is presented. The technique to prioritize the test cases works at three levels. At first level requirements are prioritized using the seven factors. At the second level the modules are prioritized using four factors. At the third level the test cases of prioritized module are further prioritized using the six factors.
The presented cost reduction framework prioritizes the requirements which are going to test in three categories. Further the categorized requirements are mapped with the past testing history of the software tested by the industry. After this testing strategies are decided which help to deliver the quality product within the lowest testing cost and time. All these techniques are explained in subsequent sections.

4.2 A MULTILEVEL SYSTEM TEST CASES PRIORITIZATION TECHNIQUE FOR OBJECT ORIENTED SOFTWARE (MSTCPTOOS)

The proposed approach works in three phases. In the first phase the requirements are prioritized. The prioritizations of requirements, modules and test cases are performed on the basis of the some factors. Every considered factor has been assigned a positive weight which is determined by using the four algorithms in SPSS. In the second phase the modules of the ordered requirement are prioritized. Finally in third phase the test cases of the particular requirement are prioritized.

4.2.1 Determination of the Weight for Considered Factors:

For determination of the contribution weight to each factor, a set of data was collected from various projects implemented by the students. The data collected from the students are analyzed by the four algorithms using SPSS Modeler [160]. The SPSS Modeler provides the strategic technique to determine the meaningful relationship among the large set of data. The SPSS Modeler has the various modeling algorithms for specific business expertise. These modeling algorithms are classification, prediction, and segmentation and association analysis. With the help of SPSS Modeler different relationships in data are investigated by applying different models. These four algorithms are the CHAID, QUEST, C 5.0 and C&R Tree [161,170,171,172]**.** The outcomes of all algorithms are analyzed and the contributions of all the considered factors are determined to decide the prediction of faults at requirement, module and test case level. The average of determined importance value obtained from all algorithms is used to prioritize the requirement, module and test cases. Figure 4.1 shows the process to find the contribution of factors to prioritize the requirements. The following steps have to be taken to apply the SPSS modeler on the set of data.

- The collected data which is going to be analyzed is imported in the SPSS modeler.
- Select the Target field from the data set, used to decide the contribution value of factors.
- Select the fields form the data set whose contribution value will be determined.
- Select the algorithm and apply it.
- Check the result and find the contribution value of the factors by analyzing the decision tree or Bar chart.

The Screen shots of the determination weight of the factors to contribute to find maximum faults are given in the Appendix E.

4.2.2 Factors Considered for Requirement Prioritization and their Reasoning

The prioritizations of the requirements are performed by using six factors. These factors are determined by the analysis of the software requirement specification. These factors are customer priority, fault proneness, requirement dependency, cost of change and risk associated with requirements. Every factor has been assigned a value between 0 to 10. The reason of using these factors are given below:

- **Customer Priority (CP):** The customer priority factor is used to determine that how much requirement is important for the customer. The customer may assign the value between the 0 to 10. The higher value shows the importance of the requirement.

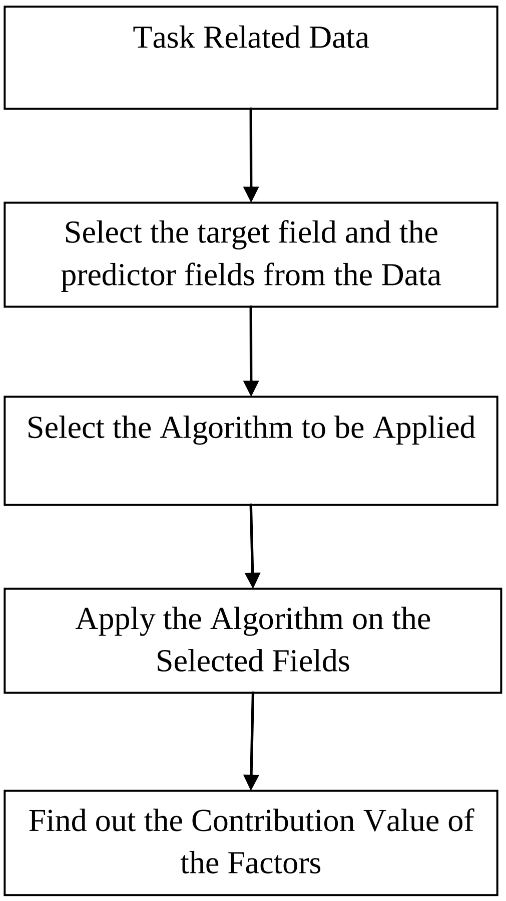

Task Related Data
Select the target field and the predictor fields from the Data
Select the Algorithm to be Applied
Apply the Algorithm on the Selected Fields
Find out the Contribution Value of the Factors

Figure 4.1 Process of Determination of the Factors Weight

- **Requirement Dependency Value (RDV):** This factor shows that how many requirements depend on a particular requirement. The value of this factor is assigned by the developer. The value of requirement dependency is assigned between the 0 and 10. This value is calculated by the following Formula 4.1

  RDV = (Number of Dependent requirements / total requirements in the whole project) *10 ---(4.1)

  The calculated RDV value of each requirement is used to assign the value in range of 0 to 10

- **Cost of Change:** This factor shows the cost of change in the requirement. The requirement may be already developed or partially developed. The requirement is volatile is nature. The requirement may be changed at later stage. So this factor shows the probability of the estimated cost if any change is introduced in the requirement at later stage. The value of this factor is assigned by the developer.

- **Implementation Complexity:** This factor shows that how much the requirement is difficult to implement. There are various ways to determine the complexity. More the complexity means more the difficulty to implement the requirement. The Value of this factor is assigned by the developer.

- **Business Impact by the Requirement:** This factor shows how much the business is affected by the requirement if it fails. There are many requirements if they are failed they don’t put impact on the business. The value of this factor is assigned by the business analytics.

- **Requirement Severity:** This factor shows concern about the security of the requirement. In a software there may be some requirement which performed some financial data and very critical for the organization. These requirements are to be protected from the transaction related to unauthorized user or other theft attack.

- **Availability of Resource:** This factor shows that whether the required resources to implement the requirement are available or not. The resources may be software tool and technology, experience developers, time, allocated budget for the particular requirement, etc.

4.2.3 Factors Considered for Prioritization of Module and their Reasoning

After prioritization of the requirements now the modules of the prioritized requirements are further prioritized. Every requirement has one or more than one modules. So these are to be prioritized. The modules are prioritized on the basis of the four factors. The values of the factors are assigned by the developer, business analytics and tester.

- **Module Dependency:** In module dependency two modules are connected in such a way that one module cannot function without other modules. If a module has high out degree of dependency then this module can impact all dependent modules. The module dependency value (MDV) can be calculated by the Formula 4.2.

  $$MDV = (NDM/TM)*10 \text{ ------------------------------------}(4.2)$$

  Where NDM is number of dependent modules and TM is the total modules presented in the requirement

- **Complexity of Module:** This factor shows how much module is complex to implement. The complexity of module can be calculated by the various ways. The value of module complexity is assigned between the 0 and 10. The value 1 shows the lowest module complexity and the value 10 shows the higher module complexity.

- **Impact on the Requirement:** This factor shows the impact of particular module failure on associated requirement. The value of factor impact on the requirement is assigned between the 0 and 10. The value of this factor is assigned by tester.

- **Requirements Coverage by Module (RCM):** This factor shows that how many requirements used a particular module. It may be possible that many requirements can use the same modules and all the requirements will not function in case of module failure. This value can be calculated by the Formula 4.3.

  RCM = (NRC/TR)*10 --(4.3)

  Where NRC is the number of requirements covered by the test cases and TR is the total numbers of the requirements in the whole software.

4.2.4 Factors Considered for Prioritization of Test Cases and their Reasoning

Every requirement has large number of test cases. It is very costly and time consuming process to execute the unordered test cases. Test case prioritization for prioritized requirements is performed on the basis of some factors. Every factor has been assigned a value between 0 to 10. The Values of factors are assigned by tester and business analytics. The reason of using these factors is given below:

- **Test Case Effectiveness:** This factor shows how much this test case is effective in past execution of the test case. The value of this factor is assigned by tester

- **Execution Frequency:** This factor shows that how frequently test cases are executed by the user. There may be some feature which was required by the user but never run by the user. So in case if they are failed, don’t put any effect on the business.

- **Test Dependency:** This factor shows the dependency of the other test cases on a particular test case. During testing some test case are never executed if some prior test cases are not executed.

- **Business Impact by Test Cases:** This factor shows the criticality of the test case for the business and shows the estimation of the business loss if the particular test case failed.

- **Feature Covered by the Test Case (FCT):** Every requirement has various features and this factor shows that how many features are covered by the test case. The value of FCT can be calculated by the Formula 4.4

  FCT = (FC/TFM)*10---(4.4)

  Where FC is the number of features covered by the test cases and TFM is total feature in associated module.

- **Fault Detection:** This factor shows the capability of detecting the maximum faults by the test case.

4.2.5 The Predicted Important Value of all the factors

Table 4.1 shows the prediction important value of each factors of the requirement from all algorithms

Table 4.1: Predicted Weight of the Factors Associated With Requirement

	CHAID	**QUEST**	**C.50**	**C&R**	**Total**	**Mean Average of the Predicted Values**
Implementation Complexity	0.04	0.01	0.2	0.12	0.37	0.0925
Cost of Change	0.07	0.09	0.15	0.04	0.35	0.0875
Business Impact	0.09	0.07	0.15	0.04	0.35	0.0875
Requirement Severity	0.12	0.09	0.07	0.01	0.29	0.0725
Requirement Dependency	0.14	0.23	0.02	0.17	0.56	0.14
Availability of Resources	0.16	0.09	0.25	0.19	0.69	0.1725
Customer Priority	0.39	0.42	0.16	0.43	1.4	0.35

Table 4.2 shows the predicted important values of each factor of the module from all algorithms

Table 4.2: Predicted Weight of the Factors Associated With Modules

	CHAID	**QUEST**	**C 5.0**	**C&R**	**Total**	**Mean Average of the Predicted Values**
Impact on Requirement	0.1	0.01	0.23	0.09	0.43	0.1075
Requirement Coverage	0.13	0.32	0.29	0.28	1.02	0.255
Complexity of Module	0.17	0.24	0.18	0.23	0.82	0.205
Module Dependency	0.59	0.42	0.31	0.4	1.72	0.43

Table 4.3 shows the predicted important values of each factor of the test cases from all algorithms

Table 4.3 Predicted Weights of the Factors Associated With Test Cases

	CHAID	**QUEST**	**C& R**	**C 5.0**	**Total**	**Mean Average of the Predicted Values**
Execution Frequency	0.09	0.2	0.07	0.19	0.55	0.1375
Feature Covered by Test Case	0.11	0.62	0.19	0.27	1.19	0.2975
Test Case Effectiveness	0.16	0.1	0.21	0.16	0.63	0.1575
Test Dependency	0.18	0.03	0.3	0.16	0.67	0.1675
Business Impact by Test Case	0.45	0.03	0.22	0.22	0.92	0.23
Fault Detection	0.01	0.03	0.02	0.01	0.07	0.0175

4.2.6 Proposed Process of Test Case Prioritization

Prioritization of the Requirement: The requirements are prioritized using the seven factors which are shown in Table 4.4. Every factor has been assigned a positive weight which shows the contribution to predict the occurrence of faults in requirements. The weight is determined by applying various data mining algorithm in SPSS modeler. The requirements are prioritized using the calculated value of requirement prioritization value (RPV) which is calculated by the Formula 4.5.

Table 4.4: Proposed Factors and Weight to Prioritize the Requirements

Proposed Factors	**Predicted Weight**
Implementation Complexity	0.0925
Cost of Change	0.0875
Business Impact	0.0875
Requirement Severity	0.0725
Requirement Dependency Value	0.14
Availability of Resources	0.1725
Customer priority	0.35

$$RPV = \sum_{i=1}^{n} W_i * V_i \text{--}(4.5)$$

where W_i is the weight of the ith factors and V_i is the value of assigned to the ith factors of requirement.

Prioritization of the Module: Modules are prioritized on the basis of the four factors. These factors are Impact on Requirement, Requirement Coverage, Complexity of Module and Module Dependency. Every factor has been assigned a positive weight which is calculated by applying the four algorithms as shown in Table

4.5. For prioritization of the modules the value of module prioritization value (MPV) is calculated by using the Formula 4.6.

Table 4.5: Proposed Factors to Prioritize the Modules

Proposed Factors	**Predicted Weight**
Impact on Requirement	0.1075
Requirement Coverage by module	0.255
Complexity of Module	0.205
Module Dependency	0.43

$$MPV = \sum_{i=1}^{n} WMF_i * VMF_i \text{--} (4.6)$$

Where WMF is the assigned weight to the ith factors and VMF is estimated value of the ith factor of module.

Prioritization of the Test Cases: In this phase the test cases of the prioritized module are prioritized. Prioritization of the test cases is performed on the basis of the six factors. These factors are the Execution Frequency, Feature covered by test case, Test Case efficiency, Test Dependency, Business Impact by test case and Fault Detection.

Every factor has assigned a positive weight which shows the capability of detection of the faults by the test cases. The weight assigned to the factors is shown in the Table 4.6. The test cases are ordered on the basis of the calculated value of the test case prioritization value (TCPV). This is calculated by the Formula 4.7.

Table 4.6: Proposed Factors to Prioritize the Test Cases

Proposed Factors	**Predicted Weight**
Execution Frequency	0.1375
Feature covered by test case	0.2975
Test Case Effectiveness	0.1575
Test Dependency	0.1675
Business Impact by test case	0.23
Fault Detection	0.0175

$$TCPV = \sum_{i=1}^{n} WTF_i * VTF_i \text{---------------------------------} (4.7)$$

Where WTF is the assigned weight to the ith factors and VTF is estimated value of the ith factor of test case.

4.2.7 Result and Analysis

For experimental verification and validation the proposed approach has been applied on two software of Inventory management implemented [155] in Java and Library information system [162] implemented in C++. The considered first software has performed various operations like addition of customer, update the customer data, add, remove, delete and update the product etc.

To analyze the effectiveness of the proposed approach some faults are introduced in the software, which are detected by applying the proposed approach. The outcomes of the proposed approach have been shown below.

The Table 4.7 shows the prioritization of the requirement

Table 4.7: Prioritization of Requirements.

	Customer	Product	Supplier	Warehouse	Sales Person	Invoice	Help	logoff	exit	Assigned weight
Customer Priority (CP).	8	8	5	7	5	8	3	5	3	0.35
Requirement dependency	8.8	8.8	5.5	5.5	5.5	3.3	0	0	0	0.14
Cost of change	8	7	5	5	5	5	0	5	0	0.0875
Implementation Complexity	8	8	5	7	5	5	0	5	0	0.0925
Business Impact by the Requirement	9	9	5	8	5	9	0	0	0	0.0875
Requirement Severity	9	9	5	7	5	5	0	0	0	0.0725
Availability of Resource	5	5	5	7	5	3	0	0	0	0.1725
RPV	7.7745	7.687	5.0825	6.72	5.0825	5.8295	1.05	3.0875	1.75	

The highest prioritized requirements has four modules, these modules are the add_edit_customer, search customer, print, and delete the customer. The prioritization process of the modules using contribution weight is shown in Table 4.8.

Table 4.8: Prioritization of Modules of Highest Prioritized Requirement

	Add_Edit_Customer	**Search**	**Delete**	**Print**
Impact on Requirement	9	7	5	2
Requirement Coverage	1.11	1.11	1.11	1.11
Complexity of Module	9	5	5	4
Module Dependency	7.5	5	0.25	0.25
MPCV	6.3205	4.2105	1.9530	1.4255

The prioritized order of the modules is Add_Edit_Customer, Search customer, Delete customer and print.

Now the test cases of the highest prioritized modules are prioritized. Table 4.9 shows the prioritization of the test cases of the add_edit_customer module.

The Table 4.9 shows the calculated value of the TCPV

Table 4.9: Prioritization of Test Cases of the Highest Prioritized Module

	TC1	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**	**Assigned Weight**
Test Case Effectiveness	0	0	0	0	0	0	0.1575
Execution Frequency	9	9	3	9	3	1	0.1375
Test Dependency	8	8	3	9	2	1	0.1675
Business Impact by Test Cases	9	9	0	9	0	1	0.23
Feature Covered by the Test Case	5	5	5	5	5	5	0.2975
Faults Detection	6	7	3	9	3	1	0.0175
TCPV	6.24	6.25	2.4	6.46	2.32	2.04	

On the basis of the obtained value of the TCPV the execution order of the test cases of the Add _Edit _Customer module is TC4, TC2, TC1, TC3, TC5, and TC6

The Graph in Figure 4.2 shows the APFD Graph of the Proposed approach, Non Prioritized approach and the PORT 2.0 Approach [136]

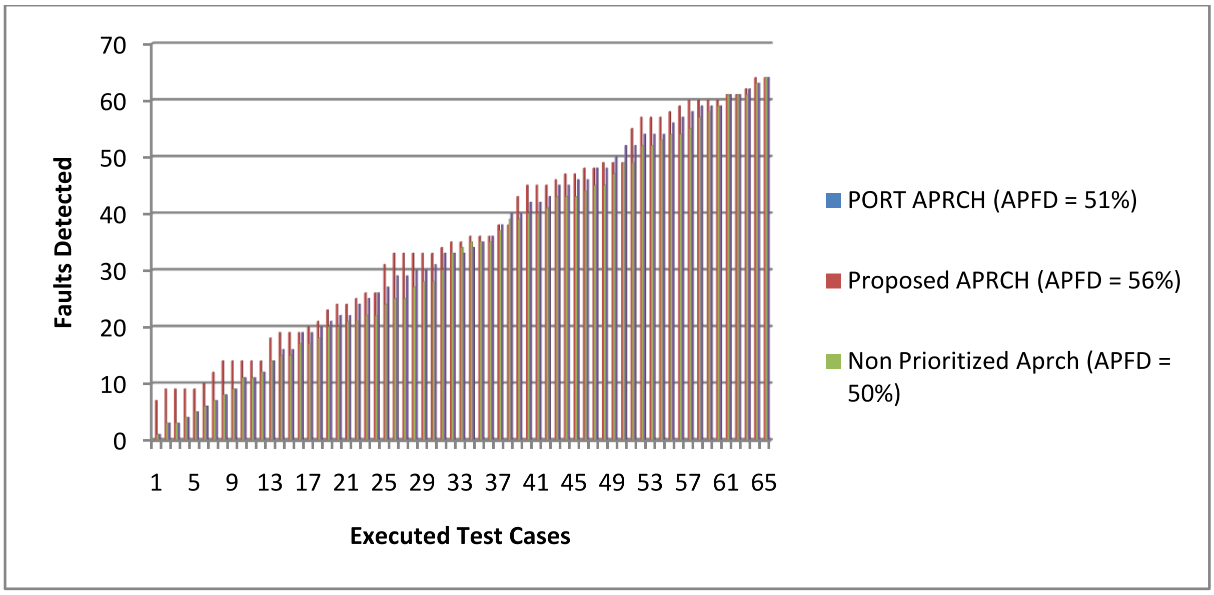


Figure 4.2: APFD Graph of non prioritized, Port and Proposed approach

APFD Value of the Non prioritized, PORT 2.0 and the Proposed approach is as shown in the Table 4.10.

Table 4.10: APFD Values of the Non Prioritized, Port and Proposed Approach

S.No	**Name of Approach**	**APFD Value (%)**
1	Non Prioritized Approach	50%
2	PORT 2.0 Approach	51%
3	Proposed Approach	56%

The same approach was applied on second considered software library information system implemented in the C++ programming language. The considered software perform the various functions like acquisition of books, membership maintenance, book issue, book return, renewal of membership, answer management queries. For experimental verification 98 faults has been introduced in the considered software which are detected using 111 test cases. The experimented results are shown in Figure 4.3.

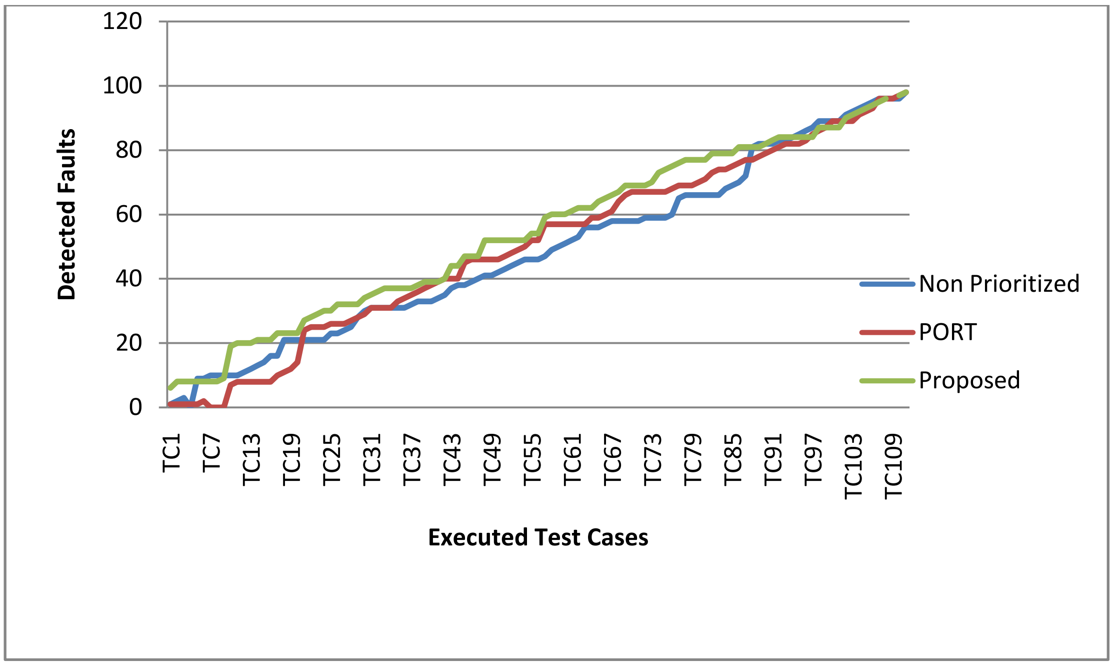


Figure 4.3: APFD Graph of Non prioritized, PORT 2.0 and Proposed approach

APFD Value of the Non prioritized, PORT 2.0 and the Proposed approach is as shown in the Table 4.11.

Table 4.11: APFD Values of the Non Prioritized, PORT 2.0 and Proposed Approach

S.No	**Name of Approach**	**APFD Value (%)**
1	Non Prioritized approach	49%
2	PORT 2.0 approach	50%
3	Proposed approach	55%

4.3 COST REDUCTION FRAMEWORK FOR OBJECT ORIENTED SYSTEM (CORFOOS)

Due to complexity of software needed to satisfy different requirements of the user, testing of software has also become quite complex. Effective software testing consumes more resources including time and increases the overall cost of software development. Various researchers have presented many techniques for reduction of testing- cost. The studies show that if the faults are not fixed in their early phase, more cost is incurred to fix the faults in the later phases. Software maintenance phase is an expensive phase as it incurs an approximate 60% of the total cost of software development.

The researchers showed that regression testing takes almost 80% of the budget allocated for testing and up to 50% of the budget for software maintenance [163]. The various constraints in software development that need to be factored in for controlling costs are budget, time, quality, risk etc.

According to finding of sixth world quality report, average spending on QA as a percentage of the total IT budget has risen from 18% in 2012 and 23% in 2013 to 26% in 2014 [164]. The share of testing budget is expected to reach 29% by 2017. Due to increase of testing cost in software development, there is a need for a technique or a framework for reduction of testing cost. With that objective, a cost reduction framework for object oriented software is presented.

4.3.1 The Proposed Framework

The proposed framework works at four levels. At the first level, requirements are analyzed and a requirement dependency graph is plotted. By using the requirement dependency graph a requirement dependency metric will be created that shows dependency value of requirements. There may be some requirements which are already implemented by the organization. At the second level, all the requirements are mapped with the past implemented requirements. After mapping, requirements will be divided into three categories: partial modified requirements, unmodified requirements and new requirements.

By using requirement dependency metric, dependency of unmodified requirement is determined. If the dependency of unmodified requirements is zero, there is no need to test them. But if the dependency value of requirement is non zero, then suitable testing strategy is required to test the requirements. The test cases are selected from the previously tested cases. In the case of partial modified requirements, an appropriate regression technique is applied to identify the affected part of requirements and for testing of requirements as a whole.

For the new requirements three models are used: Dependency model, Interaction model and Language specification model. After analysis of these models, complexity of new requirements and the faulty model of requirement are determined. By using the identified complexities and faulty model, the requirements are prioritized and suitable testing strategy is selected as shown in Figure 4.4.

4.3.2 Requirement Analysis and Requirement Dependency Graph

In this phase, an analysis of requirements is performed first of all. The analysis of requirements is performed for identifying the purpose of developing the software. After analyzing the requirements, an intermediate graph for determining the dependencies between the requirements is constructed.

In the intermediate requirement dependency graph (IRDG), the requirements are denoted by the node and dependencies between the requirements are shown by the directional edges. After constructing the IRDG, degree of each node is counted. The degree of each node is the sum of in - degree and out - degree of a node. This degree of requirements is termed as intermediate requirement dependency value (IRDV). In this way, the intermediate requirement dependency value (IRDV) metric forms using IRDG.

4.3.3 Partition of Requirements by Mapping them with Past Implemented Requirements

In this phase, the requirements are mapped with past implemented requirements. Mapping is based on functionality and implementation platform of a requirement. After mapping, the requirements are categorized as new, partially modified or unmodified.

- **New Requirements** are the emerging requirements which have not been implemented by the organization.

- **Partially Modified Requirements** are those requirements which were implemented earlier by the organization, but now there is a scope for quite a few changes.

- **Unmodified Requirements** are those requirements that are implemented without any changes.

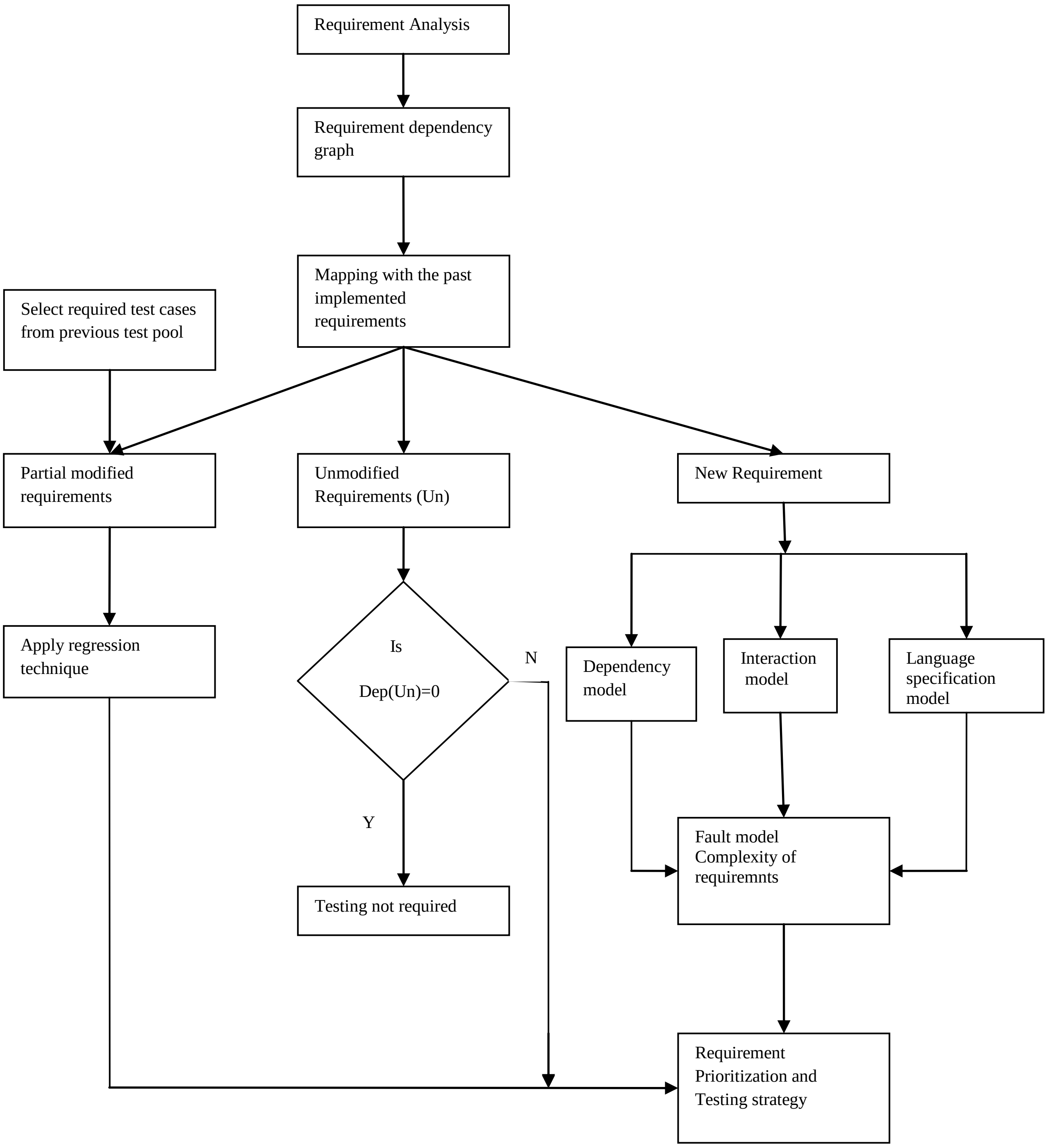

Requirement Analysis
Requirement dependency graph
Mapping with the past implemented requirements
Select required test cases from previous test pool
Partial modified requirements
Unmodified Requirements (Un)
New Requirement
Apply regression technique
Is
Dep(Un)=0
N
Y
Dependency model
Interaction model
Language specification model
Fault model Complexity of requiremnts
Testing not required
Requirement Prioritization and Testing strategy

Figure 4.4**:** Framework for Cost Reduction

Let R be a set of requirements, such that

R = {R1, R2, R3, R4, R5, R6, R7, R8, R9}

Set Pr is a set of partial modified requirements, Ur is set of unmodified requirements and Nr is the set of new requirements, such that

Pr= {R1, R4, R5}, Ur = {R5, R8, R9}, Nr = {R2, R3, R6}

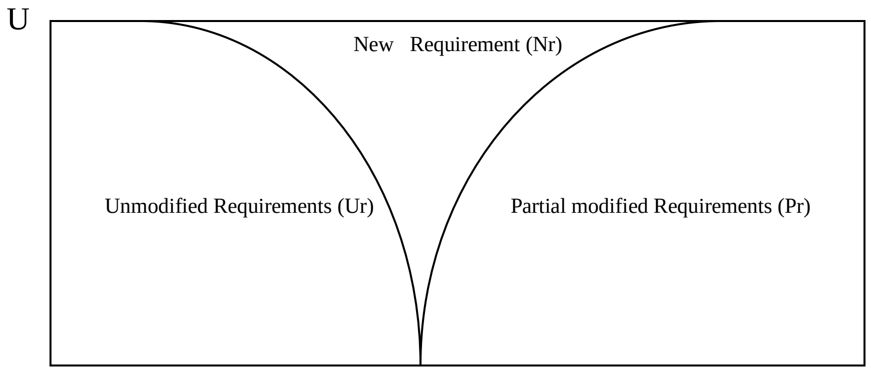

U
New Requirement (Nr)
Unmodified Requirements (Ur)
Partial modified Requirements (Pr)

Figure 4.5: Partition of the Requirements

4.3.4 Identification of Critical Requirements

By using the IRDV, any dependency of unmodified requirement is identified. If no dependency of unmodified requirement is found, then there is no need to test them. But if dependency is found, then these requirements are put in a pool of requirements to be tested and mapped with the fault model of past implemented requirements.

4.3.5 Complexity of Requirement

To find out the complexity of requirements, three models namely Dependency model, Interaction model and Language Specification model are used to calculate the testing parameters of requirements. Higher the scale of testing parameters, more are the chances of errors to occur. By using these models, the developer can identify the types of errors that might occur. Testers are able to design test cases and developer can code the requirements on the basis of these test cases as well as expected faults. This type of coding helps the developer to avoid these faults from occurring.

Dependency Model: It helps to [165,166] detect the structural dependency. Software architects always specify a set of structural constraints for the target system. Source code and related information like classes, sequence diagram and high level modules such as package and component diagram must be analyzed by the architects. Analysis of dependency includes the control dependency of the program, data dependency and dependency between the classes, method to class, method to method, polymorphism interdependency, implementation dependency, contractual dependency, dependency of program on external system call, functional dependency, etc. The control dependency covers exception handling, multithreading and synchronization. The data dependency model helps to identify the cohesion of each class and coupling between the classes which helps to determine the complexity of the requirement.

Interaction Model: Interaction model [166, 167] is used to identify the different types of interactions presented in the program. As object oriented language provides various features such as inheritance, polymorphism, message passing, and encapsulation, it is complicated and prone to errors. By using the interaction model, different types of interactions between the programs are identified. The interaction model describes the communication between the classes. The classes communicate to each other by passing messages. These messages represent the interaction between the objects. There are various types of messages in object oriented language as shown in Table 4.12.

Table 4.12: Types of Messages and Interaction in Object Oriented Programming Language.

Message	**Interaction**
Simple Message	Interaction between the classes
Synchronous Message	Interaction between the classes and interface
Asynchronous Message	Interaction between the different objects of the program
Reflexive message	Interaction between the program and native method
Return message	Interaction between the classes and distributed class

Language Specification Model: Language specification model [166,168] explains the model of the language used for implementing the requirements. It helps to identify the specified feature of language that is going to be used. Every language has a set of rules to use the various features of the language. If the specified rules for use of feature are not followed, then it will become a source of error. Using this model it helps the designer to find out which features should be used to implement the requirements for getting a quality product. The language specification model also shows which feature is prone to error and the steps to follow for using the feature in an efficient and error free manner.

Fault Model: The Fault model [166] is used to determine types of faults which are usually found during testing. The fault model shows the types of fault and reason of the faults in the software. By using the fault model, the developer or tester can analyze the software and take the required steps for reducing the faults.

Using the above mentioned models it helps to identify the testing effort of each requirement as complexity of the requirements is calculated based on them. More the complexity of the requirement more is the effort required for testing; which increases the cost of testing too. Testing effort of a requirement can be calculated by incorporating the following factors:

1. Number of classes
2. Level of inheritance
3. Number of attributes used in each class
4. Number of methods used
5. Number of native methods used
6. External system call
7. Import of the packages and API
8. Number of wrapper classes used
9. Multiple inheritance used
10. Method overloading and method overriding
11. Nested Classes

12. Expected Fault

13. Other factors

Testing effort (TE) can be calculated by the following Formula 4.8.

$$\text{Testing effort (TE)} = \sum_{i=1}^{n}(\text{fvalue}_i * \text{fweight}_i) \quad \text{-----------------------------------}(4.8)$$

where fvalue is the value assigned to the considered factors and fweight is the weight assigned to the factors, and the weight is assigned based on the criticality of the factor. Factor criticality indicates the probability of error that different factors contribute. More the factor weight more is the chances of errors to be introduced by the factors.

The requirements are prioritized and tested based on the calculated testing efforts. The value of testing efforts shows the complexity of the requirement.

4.3.6 Result and Analysis

Due to constraints of resources, the proposed approach is validated by applying it on the given requirements of a project. The requirements dependency graph is shown in Figure 4.6.

As shown in Figure 4.6, there are 10 requirements. The requirements R2 and R6 are the independent requirements. The requirements and their dependency value are shown in Table 4.13.

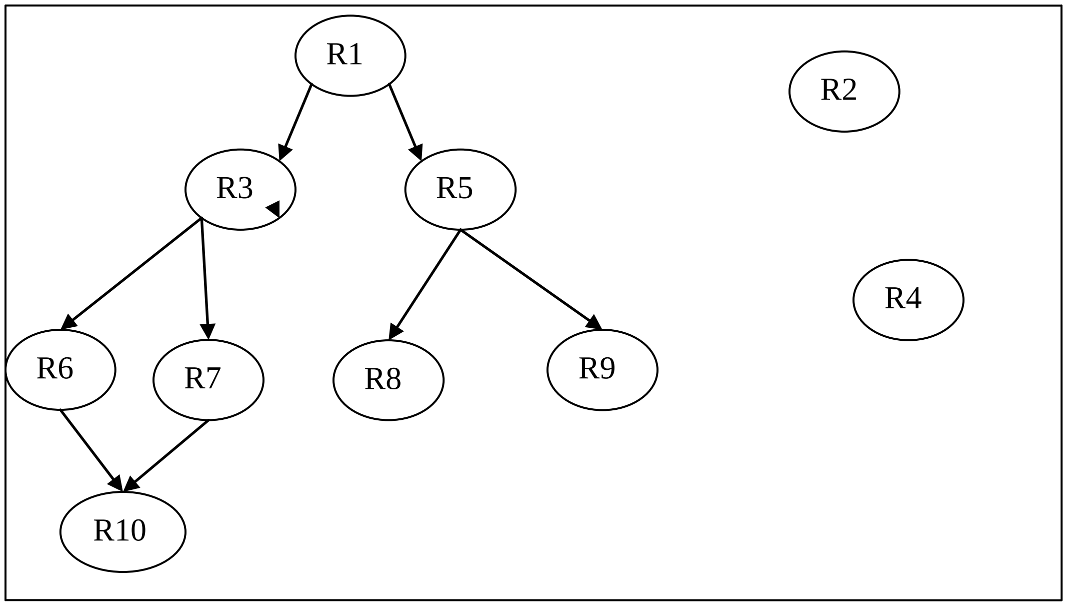


Figure 4.6: Intermediate Requirement Dependency Graph.

Table 4.13: Intermediate Requirements Dependency Value (IRDV).

S. No.	**Requirements**	**IRDV**
1	R1	7
2	R2	0
3	R3	3
4	R4	0
5	R5	2
6	R6	1
7	R7	1
8	R8	0
9	R9	0
10	R10	0

For validation of the proposed requirements, partitions of the requirements are shown below:

Un = {R2, R4}, Nr ={ R1, R3, R4, R5, R8 } Pr = {R6, R9, R10}

Since the requirement set Nr is the set of new requirements that are implemented for the first time by the organization, these should be analyzed by applying three models: interaction model, dependency model and language specification model and be prioritized accordingly.

Suppose X be the total cost to test each requirement and Y be the cost incurred in regression testing of the software. Before applying the CORFOOS, total cost to test all the requirements will be 10 X.

After applying the proposed framework, the findings are:

1. The requirements R2 and R4 are the independent requirements so the testing cost of these requirements will be zero. So there is no need to test them.
2. The requirements R6, R9, R10 are partial modified, so cost incurred to test the partial modified requirements will be 3Y.

3. The Requirements R1,R3,R4,R5,R8 are new requirements and so, their testing cost will be 5X

So, after applying CORFOOS, total cost to test all requirements of the projects are 5X + 3Y where Y < X

If we do not apply the framework proposed above, then total cost to test all the requirements will be 10 X, which is greater than the cost estimated by applying the proposed approach, which are 5X + 3Y.

4.4 CONCLUSION

In this Chapter, a Multi-level system test case prioritization technique and cost reduction framework for the object oriented software has been presented. The presented system test case prioritization technique detects the maximum faults by utilizing the less time. It will improve the quality and reduced the testing cost of the developed software. In cost reduction framework Dependency, Interaction and Fault model is used to reduce the testing cost of the software. The presented approach has been applied on software for its validity. The experimented result shows that the presented techniques and framework are very effective and helps to reduce the cost to test the software.

Chapter V

REGRESSION TEST CASE PRIORITIZATION: PROPOSED WORK

5.1 INTRODUCTION

This chapter focuses on the regression testing of object oriented systems. Regression testing is done when the system is modified to accommodate the changes. For the regression testing of the object oriented software three techniques have been presented in this chapter. In first technique Object Oriented Program Dependency Graph (OPDG) and Dynamic Slice is used to select test cases to execute the affected paths by incorporating the changes in the software. The second technique prioritizes the regression test cases on the basis of fault severity of the bugs. The third technique prioritizes the test cases using the past history of testing. The details of three techniques are given in the subsequent sections

5.2 REGRESSION TEST SELECTION FOR OBJECT ORIENTED SYSTEMS USING OPDG AND SLICING TECHNIQUE

This work is concerned about class level changes in the object oriented system and designing of algorithms to do the regression testing of system.

The regression testing in object oriented (OO) systems proceeds at two levels:

1. Application program testing
2. Interclass testing

In application program testing when the application program is modified. Regression testing is performed when application program uses modified classes.

When a Class is modified, the aim is select test cases in the class's test suite that should be re-executed. Similarly when a new class is derived from an existing class, test cases from a test suite of base class should be identified for re-execution.

The following changes are considered to do the regression testing of object oriented systems:

1. Addition of a Class
2. Deletion of a Class
3. Modification of a Class
    - Addition of a method
    - Deletion of a method
    - Modification of a method

So whenever changes occur in the OO system, either modification of an existing class will take place, or addition of a class takes place (can be a derived class or new class) in system or deletion of a class takes place. In all these cases retesting of all the classes are required which are affected by relationships like inheritance, composition and association. This will produce a large test suite. Therefore, a technique is required that will reduce the test case so that only affected classes are tested. In this direction a technique has been proposed in this chapter. An overview of proposed technique is shown in Figure 5.1.

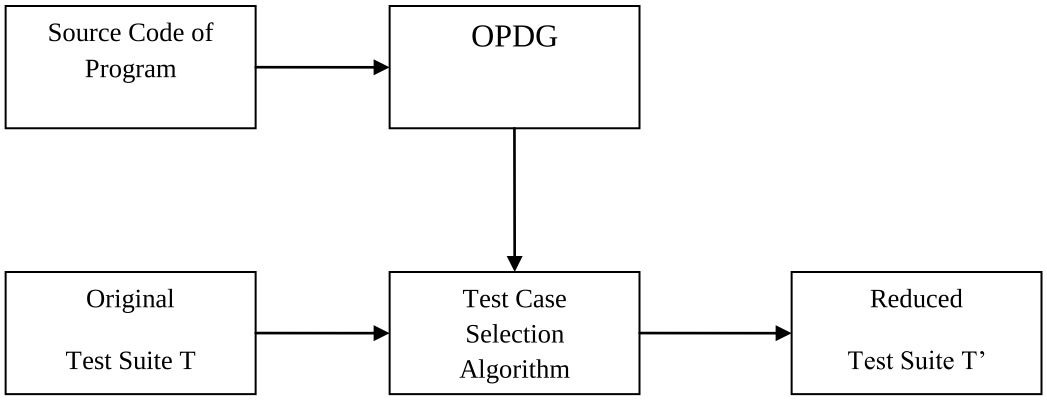


Figure 5.1 Overview of Proposed Technique

In this technique an OPDG is constructed for the modified program. In case of addition of class to the OO system, affected paths by adding the new class are identified in the OPDG and marked. Then the test cases which execute the affected path are selected for regression testing. When the class is modified, then the affected functions and affected paths are identified. Then the dynamic slicing is applied to select those test cases whose output has been affected due to modification. In case of deletion of class different cases are there, a Class Hierarchy Subgraph (CHS) is constructed to identify the class by deleting which the system will be invalid. Object oriented program dependency graph will be used to represent the object oriented programs whose regression testing is to be done. This representation is modular allowing various analysis techniques to only use the portion required for that analysis. Although this representation has three layers but in this approach only two layers CHS and CDS of OPDG will be used. CHS will be used to represent the inheritance hierarchy of classes and CDS is used to represent the control structures of various functions of classes.

5.2.1 Addition of Class

A software system always copes with requirement changes. Either a new requirement comes or an existing requirement changes. To accommodate the new requirements new classes may be added in the system. The added class may be linked to the existing classes in the system or may be an independent class in the system. If it is an independent class then there is no need of interclass testing. However if it is linked to the existing classes then interclass regression testing will play an important role.

A class may be a derived class to an existing class, or it may contain objects of other classes as its attributes i.e. a composition relationship or it may be linked by an association relationship.

The idea behind test case selection is that a new class’s methods might be calling the methods of old classes. When interclass regression testing is performed there is no need to test all the functions of old classes. Only the functions that are used by new class will be tested**.** The algorithm for selecting the test cases to test the affected classes is given in Figure 5.2.

```
INPUT: Source code of program;
Original test suite T;
OUTPUT: Reduced test suite T';
Algorithm for test case selection is:
1. Make the OPDG graph of the given classes from the source code of program.
2. After the addition of new class make the interlinked OPDG graph.
3. Mark those edges in the OPDG graph where there is a function calling dependency
between two classes.
4. Select only those test cases from the test suite T for inclusion in T' for regression testing
which execute the  marked edges.
```

Figure 5.2 : Algorithm for Selecting The Test Cases To Test The Affected Classes

5.2.2 Modification of Class

Sometimes modifications in classes have to be done to incorporate changes. These modifications in classes can be done in a variety of ways like Addition of a function in class, Modification of a function in class, Deletion of a function in class.

- **Addition of a Function in Class**

  If added function will be used by other functions in the class, then all those functions need to be tested again. Also if that function modifies the value of a variable, then dynamic slicing technique is used. All those test cases will be selected for re-execution whose outputs have been affected by the use of that variable. The algorithm for selecting the test cases for addition a function in a class the is given in Figure 5.3.

- **Modification of a Function in Class**

  This problem is called Fragile base class problem [173]. Changing the super class can affect the subclass. Modification of super class can make the subclass invalid. However functions in the subclasses and other classes can be modified but not in the super class. The algorithm for selecting the test cases for modification of a function in a class is given Figure 5.4.

INPUT: Source code of the program;

Original test suite T;

OUTPUT: Reduced test suite T';

1. From the source code of program make the OPDG of involved classes.
2. Mark the new added function.
3. If new added function modifies the definition of some variable, then trace all those statements in the program where a use of that variable has been made.
4. Mark all the affected function in the OPDG.
5. Then the dynamic slice of that variable will be computed i.e all those functions will be marked whose output may be influenced due to modification.
6. Else if it doesn't modify the value of any variable then trace those functions where that function is used (as a call to that function), then the function calling edges will be marked from that function.
7. Select those test cases which execute the marked edges and marked functions obtaining for final test suite T'

Figure 5.3: Algorithm for Selecting the Test Cases for Addition of a Function in Class

- **Deletion of a Function in Class**

  When a function is deleted then all function calls to that function will be invalid. All those functions which have used that function will be traced and stubs have to be provided for those functions calls.

5.2.3 Deletion of a Class

There are various cases of deletion of class:

- **Deletion of base class**

  When the base class is deleted then all its subclasses will be invalid. In the Figure 5.5 *Student* is the base class or super class, *Test* class is derived from the *Student* class and, from the *Test* class and *Sport* class, *Result* class is derived. When the super class *Student* is deleted, test class and *Result* class will be invalid. However *Sport* class continues to work, because it has no relation to the *Student* class. Thus it can be concluded that

```
INPUT: Source code of the program;
          Original test suite T;
OUTPUT: Selected test suite T';
Algorithm for test case selection is:
1. From the source code of program make the OPDG of involved classes.
2. Mark the statement in the OPDG which has been modified.
3. Also mark the function in the OPDG which has been modified.
4. Mark all the affected functions due to that modification in the OPDG.
5. If that function is used in other functions, then mark those function calling edges in the
   OPDG.
6. Compute the dynamic slice of the changed variable in the modified statement i.e  mark
   those functions whose output have been influenced due to modification.
7. Select those test cases for regression testing which executes those marked edges and
   marked functions obtaining for final test suite T'.
```

Figure 5.4: Algorithm for Selecting the Test Cases for Modification of a Function in Class

- A base class can’t be deleted because then all its subclasses will become invalid.

- If the requirement of base class finishes then base class should be changed to abstract class.

- By changing that class into abstract class its subclasses will remain valid but application program can’t instantiate that class

- **Deletion of Derived Class**

  When the derived class is deleted, then the base class will not have any effect. However if a derived class is used by another class and derived class is deleted, then that class will be invalid.

In the Figure 5.5 when the class *Result* is deleted, links to *put_number* and *put_marks* will be deleted and it will have no effect on its super class *Student* and *Test*.

However when the class *Test* is deleted, class *Result* will be invalid. When class *Sport* is deleted a dummy function call for the function *put_score* has to be provided. Thus it can be concluded that

- If the requirement of derived class finishes then a derived class can be deleted and its super class will not have any effect of its deletion.

- However if that class is used by other classes then those classes will become invalid.

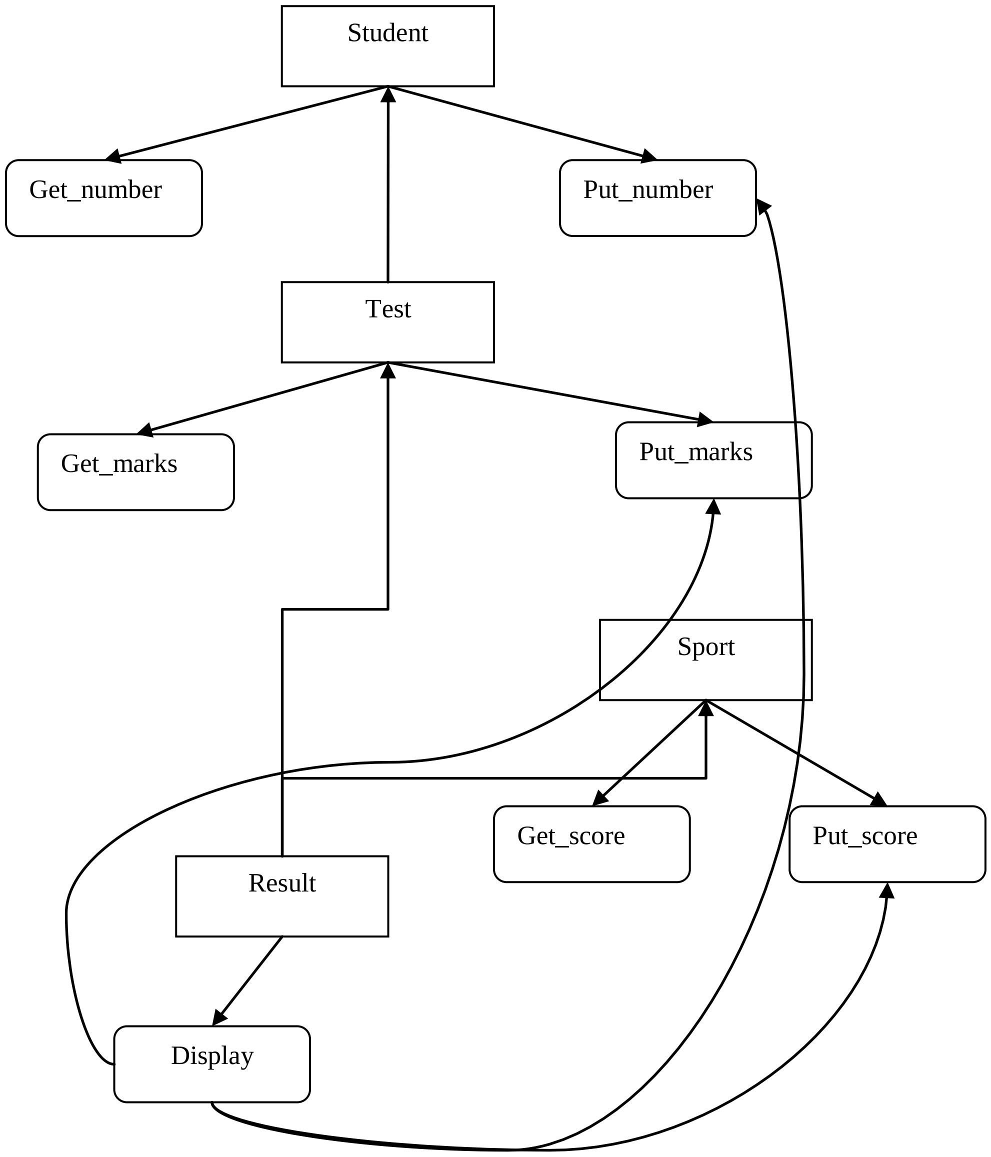


Figure 5.5: Deletion of a Class

5.2.4 Effectiveness of Proposed Approach

To analyze the effectiveness of proposed approach it is applied to various case studies. Different programming examples of C++ are considered for each case i.e for addition of class, modification of class, deletion of class. Then the OPDG for each case study is constructed, subsequent test cases are designed and then the proposed approach is applied. Finally on the basis of affected paths test cases are selected.

Addition of Class

To demonstrate the selection of test cases through the proposed approach, it is applied to an C++ programming case study. The considered case study performs simple banking operations like depositing the amount, withdrawing the amount, computing the interest on deposited amount and displaying the balance. In this case study it is shown the addition of class through inheritance. A base class *Account* is there then a class *SAccount* (SavingsAccount) is derived. The class *SAccount* uses the functions of *Account* class. The code of program is given below.

Source Code of the Considered Case study

```
class Account
{
protected:
char name [20];
int ano;
char type;
double acc_balance;
public:
account( ) {};
void credit( double );
void debit( double );
double getbalance( );
};
Account :: Account ( double initialdeposit, char *aname, int no, char atype )
{
c1p1 if( initialdeposit < 0.0 )
s1 cout<<" Invalid entry";
s2 sacc_balance = 0;
else
s3 acc_balance = initialdeposit;
s4 strcpy ( name, aname );
s5 ano = no;
s6 type = atype;
}
void Account :: credit ( double deposit )
{
s7 acc_balance += deposit;
s8 cout<<" Credited to account";
}
bool Account :: debit ( double withdraw )
{
c2p2 if( withdraw > acc_balance )
{
s9 cout<<" Invalid entry";
s10 return false;
}
else
{
s11 acct_balance -= withdraw;
s12 cout<< "Debited to account";
s13 return true;
}
}
```

```
double Account :: getbalance ( )
{
s14 cout<<" Current account balance is:"<< acc_balance;
s15 return acc_balance;
}
class SAccount :: public Account
{
private:
double savingBalance;
double earnedInterest;
double annual_interestrate;
public:
SAccount( double );
~SAccount( );
void calcInterest( );
void modinterest(double );
void print( );
};
SAccount :: SAccount( double savDeposit ) : Account( savDeposit )
{
C3P3 if( savDeposit >=0 )
{
S16 savingBalance = savDeposit;
}
else
{
S17 savingBalance = 0;
S18 cout<<" Invalid entry";
}
}
SAccount :: ~SAccount( )
```

```
{
}
void SAccount :: modInterest( double newrate )
{
 S19annual_interestrate = newrate;
}
void SAccount :: calcInterest( )
{
S20 earnedInterest = ( Account :: acc_balance * annual_interestrate );
S21 credit( earnedInteresr );
}
void SAccount :: print( )
{
S22 cout<<"Balance is: "<< Account :: getbalance( );
}

int main( )
{
while(1)
{
char c;
char acname[30];
int acno;
char atype;
cout<<"\n Account opening system:";
cout<<"\n Enter the customer name";
for( int i=0; ( i=getche( ) )! = '\0'; i++)
acname[i] = c;
acname[i] = '\0';
cout<<"\n Enter the account number";
cin>>acno;
```

```
int ch;
while( 1 )
{
cout<<"\n Enter the account type: 's' for savings and 'c' for checking";
cin>>atype;
if(atype == 's')
{
SAccount s1(100) :: account( acname, acno, atype );
while (1)
{
cout<<"\n Saving Account menu";
cout<<"\n 1. Deposit";
cout<<"\n 2. Withdraw";
cout<<"\n 3. Compute interest";
cout<<"\n 4. Display balance";
cout<<"\n 5. Exit";
cout<<"\n Enter your choice";
cin>>ch;
if(ch == 1)
s1.credit( 2500 );
if(ch == 2)
s1.debit( 500 );
if( ch == 3 )
s1.modifyinterest( );
s1.calculateinterest( );
if( ch == 4 )
s1.print( );
}
else
{
cout<<"\n Invalid choice";
}
continue;
}
}
```

Construction of OPDG : An OPDG graph of considered case study is constructed to identify the affected paths for interclass testing. The edges containing the affected paths in the OPDG graph are colored red for distinguishing those edges from other edges. The OPDG graph of this example is shown in the Figure 5.6.

Designing of Test Cases

On adding the class SAccount through inheritance to the base class *Account,* interclasss testing is to be performed such that total test cases are run to test both classes on adding the class as shown in the Table 5.1.

Table 5.1: Test Cases Designed for Addition of Class

Inputs	**Customer Name**	**Atype**	**Ch**
TestCase 1	Ram	S	1
TestCase 2	Ram	S	2
TestCase 3	Ram	S	3
TestCase 4	Ram	S	4
TestCase 5	Ram	S	5

But if test cases are to be selected for interclass testing, while considering the affected path in the OPDG, only test case 3 and test case 4 execute the affected path. Test cases selected for execution are shown in the Table 5.2

Table 5.2 Test Cases Selected for Addition of Class

Inputs	CustomerName	Atype	Ch
TestCase 3	Ram	S	3
TestCase 4	Ram	S	4

So while performing the interclass testing on adding the class *SAccount*, only test case 3 and test case 4 need to be selected for re- execution instead of re-executing all test cases.

Modification of Class

Modification of a class can be done in three ways:

1. Adding a new function in the class
2. Modifying the existing function
3. Deleting the function in class.

All three cases are considered with a programming example.

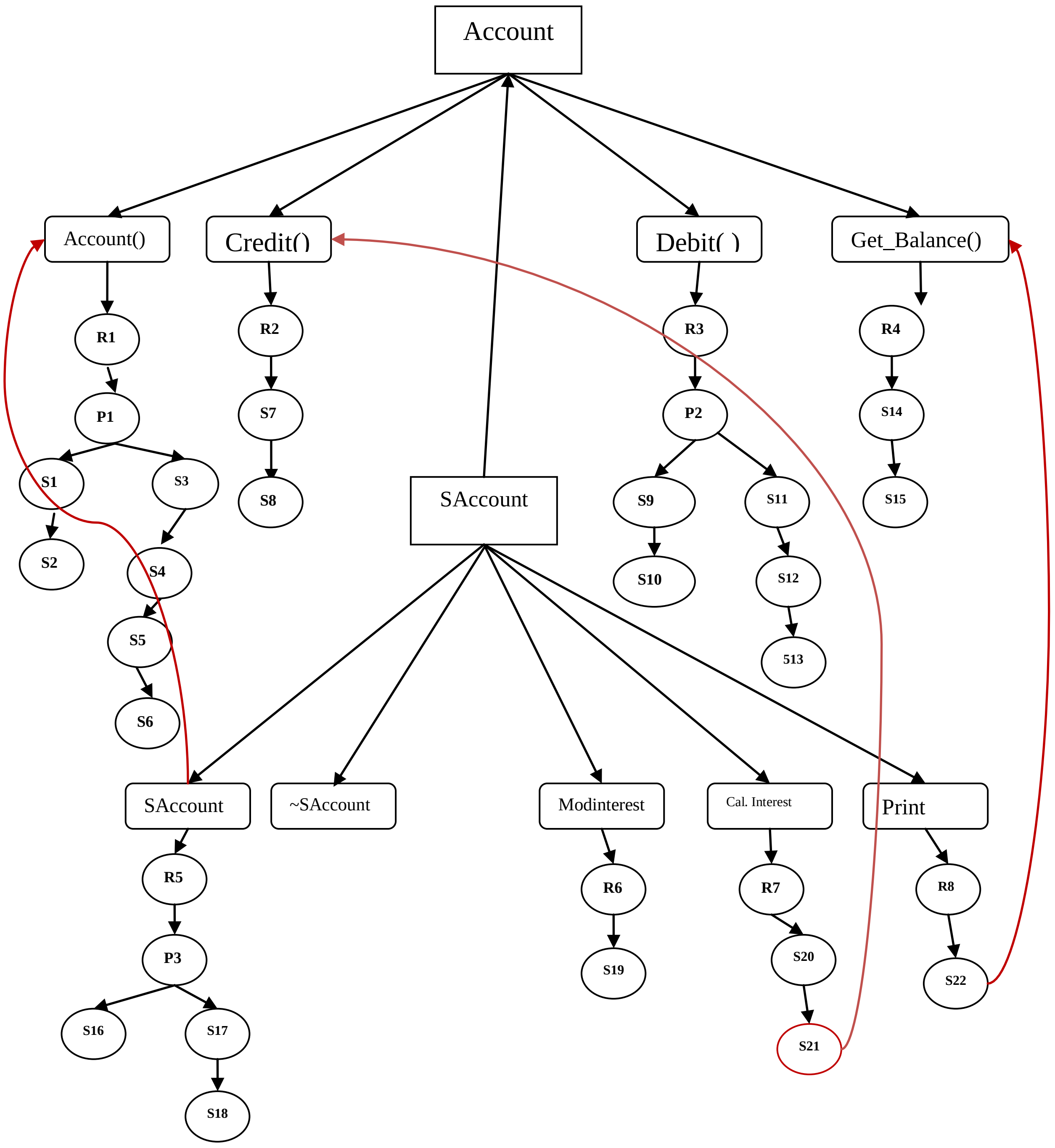


Figure 5.6: OPDG for Addition of Class

Modification of a function in the class

The technique to select test cases in the proposed approach is applied to a programming example of C++ to show the selection of test cases. The case study takes computes the simple interest and compound interest by taking the inputs of present value, rate and time. There is a function *modify* in the class *Interest*, that does the task of modifying the interest rates. This function has been modified. This function prior to modification modifies the value of interest rate to compute the simple interest. Later a statement has been added in the modified function. This statement modifies the value of interest rate to compute the compound interest. The code of program prior to modification is shown below:

Source code for the consider Case Study

```
class Interest
{
protected:
double  r;
double cr;
public:
Interest( );
void modify( );
{
S1 r = r + (r* 0.1);
}
};
class SInterest :: public Interest
{
```

```
protected:
int p,t;
double si;
public:
SInterest( int x, double y, int z )
{
S2 p = x;
S3 r = y;
S4 t = z;
}
void cal_interest( )
{
S5 si = ( p*r*t) / 100;
}
void print( )
{
S6 cout<<” Interest is: “<< si;
}
class CInterest :: public interest
{
protected:
double  ci;
int p, t;
public:
CInterest ( int x, double y, int z )
{
S7 p = x;
S8 cr = y;
S9 t = z;
}
void cal_interest( )
```

```
{
S10 ci = p* pow((1+cr/100), t);
}
void print( )
{
S11 cout<< " Interest is: "<<ci;
}
};

int main( )
{
int  p, t;
double  r, cr;
while( 1 )
{
Interest iob;
cout<<" Enter the values of  p,r,cr and t;
cin>> p>>r>>cr>>t;
if( t == 0 )
break;
if( t == 1)
iob = SInterest( p, r, t );
else
iob = CInterest( p, r, t );
iob.modify( );
if( p > 1000)
{
iob.calculateinterest( );
iob.print( );
}
}
```

After modification of function modify in class Interest, the code of program is shown below:

```
class Interest
{
protected:
double  r;
double cr;
public:
Interest( );
void modify( ) // modified function
{
S1 r = r + (r* 0.3);
S12 cr = cr + (cr*0.3);
cout<<" Executed modify";
}
};
class SInterest :: public Interest
{
protected:
int p,t;
double si;
public:
SInterest( int x, double y, int z )
{
S2 p = x;
S3 r = y;
S4 t = z;
}
void cal_interest( )
{
```

```
S5 si = ( p*r*t) / 100;
}
void print( )
{
S6 cout<<" Simple Interest is: "<< si;
}
class CInterest :: public interest
{
protected:
double  ci;
int p, t;
public:
CInterest ( int x, double y, int z )
{
S7 p = x;
S8 cr = y;
S9 t = z;
}
void cal_interest( )
{
S10 ci = p*(1+cr/100)^t;
}
void print( )
{
S11 cout<< " Compound Interest is: "<<ci;
}
};
 int main( )
{
int  p, t;
double  r, cr;
```

```
while( 1 )
{
cout<<" Enter the values of  p,r,cr and t;
cin>> p>>r>>cr>>t;
if( t == 0 )
{
Interest iob;
iob.modify( );
break;
}
elseif( t == 1)
{
SInterest s1( p, r, t );
s1.modify( );
s1.calculateinterest( );
s1.print( );
break;
}
elseif(t == 2)
{
CInterest c1( p, r, t );
c1.modify( );
c1.calculateinterest( );
c1.print( );
}
}
```

Construction of OPDG: In the Figure 5.7 OPDG of the modified program is constructed. The modified statement in the OPDG is colored red and also all other statements which have been affected due to modification are also colored red.

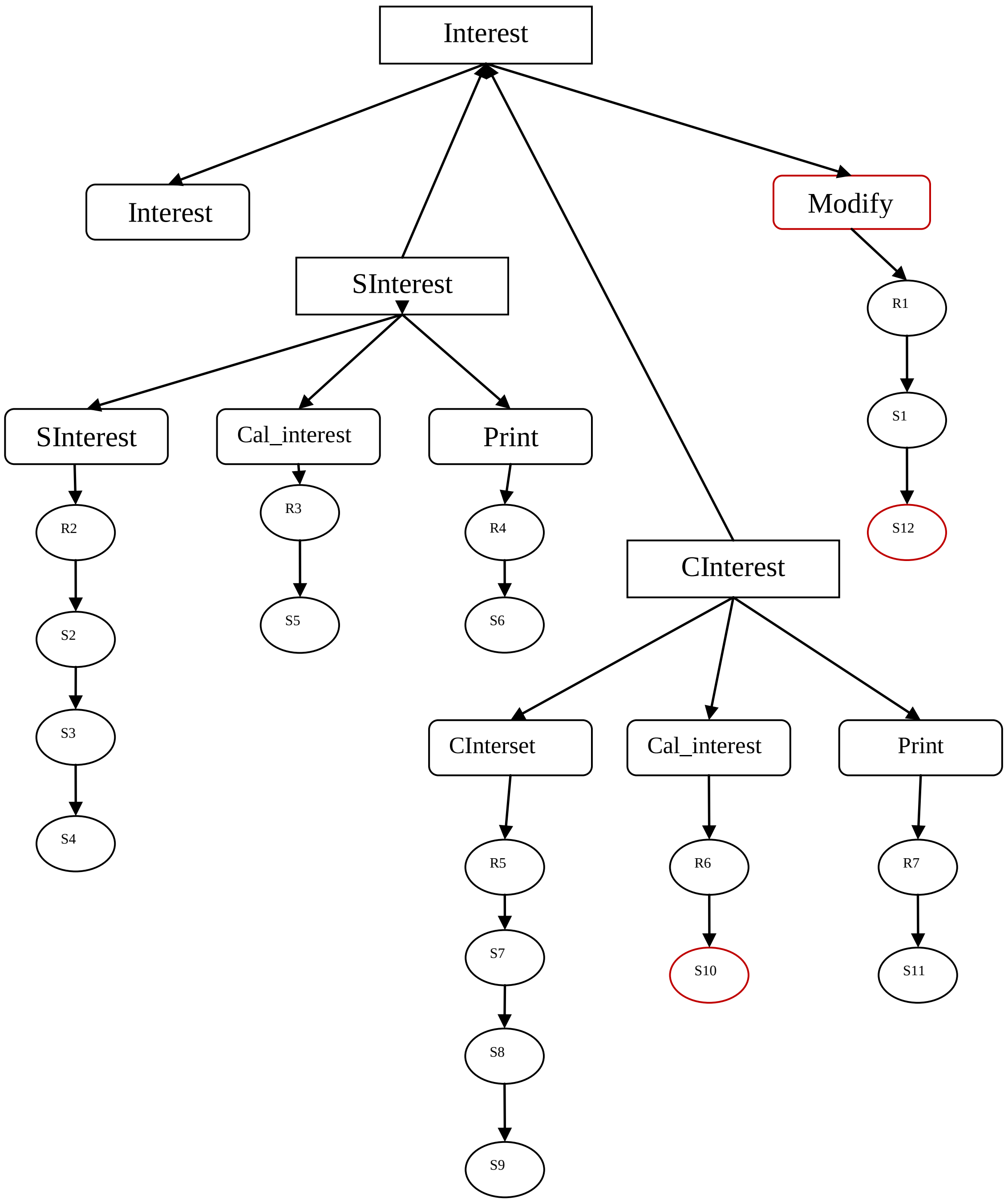


Figure 5.7: OPDG for Modification of a Function in a Class

Designing of test cases

Each test case executes a sequence of function calls according to the input provided. The total number of test cases according to this application program is three. By applying the proposed approach only TestCase3 executes the affected function (Cal_interest of class CInterest) have been affected i.e the statements executed by TestCase3 comes under the dynamic slice of the modified function. Although all the three test cases execute the modified function, but the TestCase1 and TestCase2 don't have any effect on their output. Total number of test cases is shown in the Table 5.3.

Table 5.3: Test Cases Designed Before the Modification of a Function in a Class

Inputs	**P**	**R**	**cr**	**t**	**Output**
TestCase1	1500	1.5	1.5	0	Executed modify
TestCase2	1500	1.5	1.5	1	Executed modify SimpleInterest is:29.25
TestCase3	1500	1.5	1.5	2	Executed modify CompoundInterest is:1983.75

Table 5.4: Test Cases After the Modification of a Function in a Class

Inputs	**P**	**R**	**Cr**	**t**	**Output**
TestCase1	1500	1.5	1.5	0	Executed modify
TestCase2	1500	1.5	1.5	1	Executed modify SimpleInterest is:29.25
TestCase3	1500	1.5	1.5	2	Executed modify CompoundInterest is:2142.03

Test case selected for re-execution is shown in the Table 5.5.

Table 5.5: Test Case Selected for Re Execution for Modification of a Function

Inputs	**P**	**R**	**Cr**	**T**	**Output**
TestCase3	1500	1.5	1.5	2	Executed modify Compound Interest is:2142.03

Only TestCase3 needs to be rerun, because the dynamic slice of the variable with respect to modified variable ‘cr’ comes under the TestCase3. When TestCase1 and TestCase2 are executed they have no effect on the output. They don’t execute the functions which can have effect on output due to modification.

Addition of a Function in Class

During the modification of a class a function may be added in the class. To apply the proposed approach the earlier example of class *Account* and *SAccount* is considered. In this example a new function called *getbonus( )* is added in the class *SAccount*. This function *getbonus( )* calculates the bonus given to the accountholder and credits that bonus to his account and also modifies the value of variable called annualInterestRate. The code of program is given below.

Source code for consider Case Study

```
Class Account
{
protected:
char name [20];
int ano;
char type;
double acc_balance;
public:
account( ) {};
void credit( double );
void debit( double );
double getbalance( );
};
```

```
Account :: Account ( double initialdeposit, char *aname, int no, char atype )
{
c1p1 if( initialdeposit < 0.0 )
s1 cout<<" Invalid entry";
s2 acc_balance = 0;
else
s3 acc_balance = initialdeposit;
s4 strcpy ( name, aname );
s5 ano = no;
s6 type = atype;
}
void Account :: credit ( double deposit )
{
s7 acc_balance += deposit;
s8 cout<<" Credited to account";
}
bool Account :: debit ( double withdraw )
{
c2p2 if( withdraw > acc_balance )
{
s9 cout<<" Invalid entry";
s10 return false;
}
else
{
s11 acct_balance -= withdraw;
s12 cout<< "Debited to account";
s13 return true;
}
}
double Account :: getbalance ( )
```

```
{
s14 cout<<" Current account balance is:"<< acc_balance;
s15 return acc_balance;
}
class SAccount :: public Account
{
private:
double savingBalance;
double earnedInterest;
double annual_interestrate;
double bonus;
public:
SAccount( double );
~SAccount( );
void calcInterest( );
void print( );
void modinterest( double );
void getbonus( );
};
SAccount :: SAccount( double savDeposit, double interestrate ) : Account(
savDeposit, char *aname, int no, char atype )
{
C3P3 if( savDeposit >=0 )
{
S16 savingBalance = savDeposit;
}
else
{
S17 savingBalance = 0;
S18 cout<<" Invalid entry";
}
```

```
S26 annualInterestRate = interestrate;
}
SAccount :: ~SAccount( )
{
}
void SAccount :: modinterest( double newrate )
{
 S19annual_interestrate = newrate;
}
void SAccount :: calcInterest( )
{
S20 earnedInterest = ( Account :: acc_balance * annual_interestrate );
S21 credit( earnedInteresr );
}
void SAccount :: print( )
{
S22 cout<<”Balance is: “<< Account :: getbalance( );
}
void SAccount :: getbonus( )
{
S23 bonus =( Account :: acc_balance * 10) / 100;
S24 credit (bonus);
S25 annualInterestrate = annualInterestrate – (annualInterest * 0.05 ) / 100;
}
int main( )
 {
char c;
char acname[30];
int acno;
char atype;
```

```
double ir;
cout<<"\n Account opening system:";
cout<<"\n Enter the customer name";
for( int i=0; ( i=getche( ) )! = '\0'; i++)
acname[i] = c;
acname[i] = '\0';
cout<<"\n Enter the account number";
cin>>acno;
cout<<"\n Enter the account type: 's' for savings and 'c' for checking";
cin>>atype;
cout<<"\n Enter the value of annual interest rate";
cin>>ir;
SAccount s1(100,ir) :: Account( acname, acno, atype );
 while (s1.atype == 's')
 {
while (1)
{
cout<<"\n Saving Account menu";
cout<<"\n 1. Deposit";
cout<<"\n 2. Withdraw";
cout<<"\n 3. Compute interest";
cout<<"\n 4. Get bonus";
cout<<"\n 5. Get bonus and modify interest rate";
cout<<"\n 6. Get bonus and compute interest";
cout<<"\n 7. Display balance";
cout<<"\n 8. Exit";
cout<<"\n Enter your choice";
cin>>ch;
if(ch == 1)
s1.credit( 2500 );
if(ch == 2)
s1.debit( 500 );
if( ch == 3 )
s1.calculateinterest( );
if( ch == 4 )
s1.getbonus( );
if( ch == 5)
{
s1.getbonus( );
s1.calculateinterest( )’
}
if( ch == 6 )
{
s1.getbonus( );
s1.modinterest( 45 );
}
if( ch == 7 )
s1.print( );
else
{
cout<<”\n Invalid choice”;
}
continue;
}
}
```

Construction of OPDG : The OPDG graph of the above program is constructed in the Figure 5.8. The new added function getbonus() in the class SAccount is colored red. Also the statement in the new function which can have effect on the other functions is also colored red.

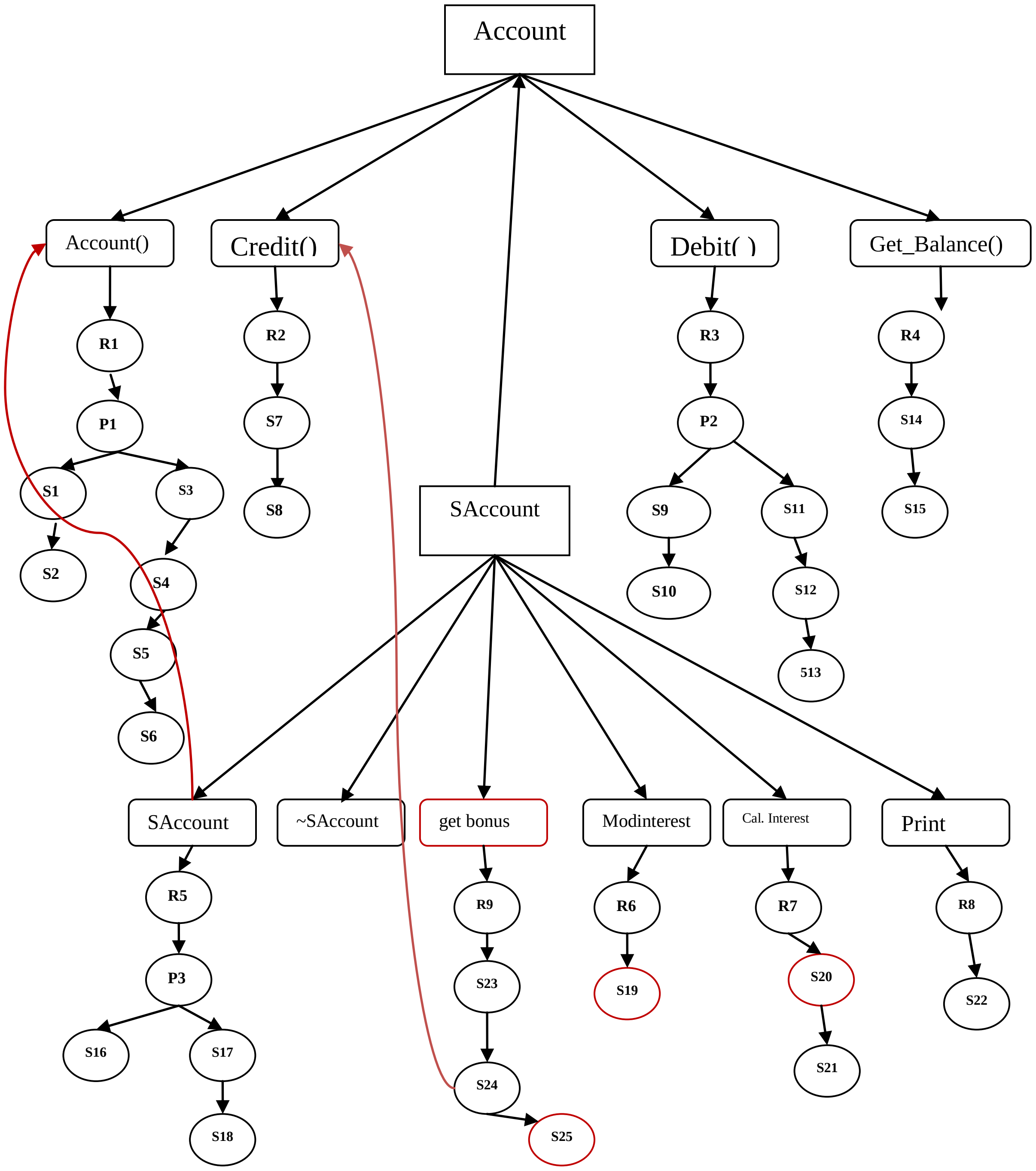


Figure 5.8: OPDG for Addition of a Function in a Class

Designing of Test Cases

In this example in the class *SAccount* a new function *getbonus()* has been added. This function *getbonus()* computes the bonus. This function makes a call to the function *credit()*. And this function also modifies the value of a variable called *annualInterestRate*. So affected path will be identified and also slicing will be done based on variable *annualInterestRate*. So all those functions will be identified where a use of variable *annualInterestRate* has been done and output has been affected in that function due to modification. The variable *annualInterestRate* which is given new definition in the function *getbonus()* is used in the functions *cal_interest()* and *mod_interest()*. But only the function *cal_interest()* need to be tested in combination with *getbonus()* , because in the function *cal_interest()*, the modified variable is used to compute the interest, thus output have been affected.Thus function *cal_interest()* comes under the dynamic slice of new function. But in the function *mod_interest()* only a new definition is given to that variable, so there is no need to run that function in combination with *mod_interest()*. Total test cases run to test the program are shown in the Table 5.6.

Table 5.6: Test Cases Designed for Addition of a Function in a Class

Inputs	**savingBalance**	**Acname**	**Acno**	**Atype**	**ir**	**ch**
TestCase 1	500	Ram	1261	S	1.5	1
TestCase 2	2000	Ram	1262	S	1.5	2
TestCase 3	4000	Ram	1231	S	1.6	3
TestCase 4	5000	Ram	1263	S	1.5	4
TestCase 5	2000	Ram	1261	S	3.5	5
TestCase 6	3000	Ram	1261	S	1.4	6
TestCase 7	3000	Ram	1264	S	1.2	7

Test cases that are selected are shown below in the Table 5.7.

Table 5.7: Test Cases Selected for Re-execution for Addition of a Function in a Class

Inputs	**savingBalance**	**Acname**	**acno**	**Atype**	**Ir**	**Ch**
TestCase 4	5000	Ram	1263	S	1.5	4
TestCase 5	2000	Ram	1261	S	3.5	5

5.2.5 Analysis of Proposed Approach

The Table 5.8 shows that there is significant percentage of reduction of test cases. The percentage of reduction of test cases shows the effectiveness of proposed approach.

Table 5.8: Analysis of Proposed Approach

S.No.	**Programe Name**	**No. Of test cases**	**No. Of selected test cases**	**% of reduction of test cases**
1	Addition of class	5	2	60%
2	Modification of function in a class	3	1	66.6%
3	Addition of a function in a class	7	2	71.42 %

5.3 A FAULT – SEVERITY BASED REGRESSION TEST CASE PRIORITIZATION TECHNIQUE FOR OBJECT ORIENTED SOFTWARE (FSRTCPTOOS)

In this section, a regression test case prioritization technique for object oriented programs is proposed. One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time. Inheritance makes the subclasses dependent on the super class and a change in the super class will directly affect the subclasses that inherit from it means retesting of all subclasses is required.

The probability of error propagation in inheritance hierarchy depends on the number of inherited attributes/methods, level of class in inheritance hierarchy and the number of descendent classes. So, the first level prioritization involves prioritizing the classes depending on the number of descendents of that class, number of inherited attributes/methods and level of the class in inheritance hierarchy. The proposed work includes two level prioritization, in which the first level prioritization involves prioritizing the classes using inheritance hierarchy whereas the second level prioritization involves prioritizing the test cases of each class.

If number of levels are less than or equal to 3, the testing effort can be calculated as:

*Testing effort = (number of descendents + number of inherited attributes/methods) * (4 -level) --(5.1)*

If number of levels are greater than 3, the testing effort can be calculated as:

*Testing effort = (number of descendents + number of inherited attributes/methods) * (level - 3)--(5.2)*

The base class at level 1 of inheritance hierarchy is always assigned highest priority. If any error get propagated from this class, will affect the entire hierarchy, because all the classes below this level will inherit the properties of base class. The second level of prioritization is the ordering of test cases of each selected class and it is done by technique fault coverage per unit time taken. Every test case is designed when program is developed. The test case is stored with time taken by it and number of faults detected by it. Each fault is assigned a weight on the basis of its criticality.

5.3.1 First Level Prioritization

The first level prioritization technique prioritizes the classes of object oriented software using inheritance hierarchy. In inheritance hierarchy the classes at lower level inherits the properties of classes at upper level. Hence, the derived classes are dependent on the base classes, This dependency increases the probability of error propagation through the inheritance hierarchy. Hence the classes should be tested in such an order that the classes with higher probability of error propagation get tested first.

The technique for prioritizing the classes of object oriented software has been proposed to find faults quickly. The probability of error propagation in inheritance hierarchy depends on the number of inherited attributes/methods, level of class in inheritance hierarchy and the number of descendent classes. The base class should be assigned the highest priority because if any errors get propagated from this class, will affect the entire hierarchy. So the classes should be ordered in such a way that error propagation can be minimized.

The classes at lower level are assigned priority based on the level of class in inheritance hierarchy, number of inherited attributes and number of descendent classes.
An algorithm has been proposed for prioritizing the classes of object oriented software using inheritance hierarchy. The classes of inheritance hierarchy have inherent complex relationships due to the dependency of derived classes over subclasses. This algorithm prioritizes the classes in such a way so that faults could be found earlier and the probability of error propagation through the inheritance hierarchy could be minimized. The algorithm in Figure 5.9 describes the technique used for first level prioritization:

```
First_ level_ prioritization (P, n)
Where P is complete program and n is the number of levels in inheritance hierarchy.
Begin
1. Assign level number to each class in the inheritance hierarchy.
2. Assign highest priority to the base class at level one of the hierarchy.
3. For (level=2; level<=n; level++)
        a) Find number of descendents for each class.
        b) Find number of inherited attributes/methods for each class.
        c) If no of levels is less than or equal to 3,then
                Testing effort = (no. of descendents + no of inherited attributes/methods) * (4 - level)
           Else
                Testing effort = (no. of descendents + no of inherited attributes/methods) * (level - 3)
d) Assign priority to each class depending on the value of testing effort.
      (highest testing effort value gets the highest priority)
end
```

Figure 5.9: Algorithm for First Level Prioritization

5.3.2 Second Level Prioritization

Based on first level prioritization for prioritizing the classes of object oriented software, now second level prioritization has been proposed so that test cases of each class can be prioritized.

Second level prioritization is a technique to prioritize test cases on the bases of fault coverage per unit time.

The classes are prioritized using first level prioritization are input to the second level prioritization where the test cases of each individual classes are prioritized. The test cases are prioritized based on fault weight and fault coverage. The fault weights are assigned based on severity and coverage is based on number of faults found by particular test case in per unit time.

The test cases that detect faults which have not been discovered earlier and are more critical are prioritized first. The algorithm in Figure 5.10 explains the second level prioritization used for ordering the test case of each particular class of inheritance heirarchy.

```
//there are M test cases and N faults and each fault is assigned some weight.
Begin
1. T is original test suite, T' is prioritized test suite
2.  Calculate fault_weight per unit time  by each test case.
3. Arrange them in decreasing order.
4. Remove the best one from T and add it to T'.
5. while(T! empty)
begin
6. Calculate weight of new faults detected per unit time of each test case.
// New Fault means those fault which are not detected by any test case in T'.
7. Remove the best one from T and add it to T'
8. Go to step 5.
end
9. Return T'.
End
```

Figure 5.10: Algorithm for Second Level Prioritization

5.3.3 Proposed Fault Table (FSRTCPTOOS)

Faults can be categorized on the basis of severity, and assigned a weight on the basis of structure of program. Weights of faults are shown in Table 5.9.

Table 5.9: General Fault Weight Table

Type of fault	**Fault weight**
Type mismatch of arguments in function	2
Check condition in if block	2
Fault in Statements inside if block	1
Fault in switch statement	2
Fault in for loop	3
Fault in recursion	4
Fault in do while loop	2
Condition statement under condition statement	4
Loop under condition statement	3
Fault in nested loop	4
Lack of memory	3
Improper use of access specifier	3
External function not called properly	2
Improper Type casting	3
Exception handling problem	2
Method signature problem	2

5.3.4 Experimental Evaluation and Analysis of Proposed Work

In this section the proposed technique has been verified and analyzed by taking a case study of student. Case *Study* consists of four classes, study, *Lec_time*, *Sports_time* and *Usetime*. The class *Study* inherits two classes, *Lec_time* and *Sports_time* and the *Lec_time* further inherits *Usetime*. The testing effort has been calculated by using number of descendents, number of inherited attributes/ methods and the level of a class in inheritance hierarchy.

Considered Case study

The inheritance hierarchy shown in Figure 5.11 has been used to analyze the proposed technique.

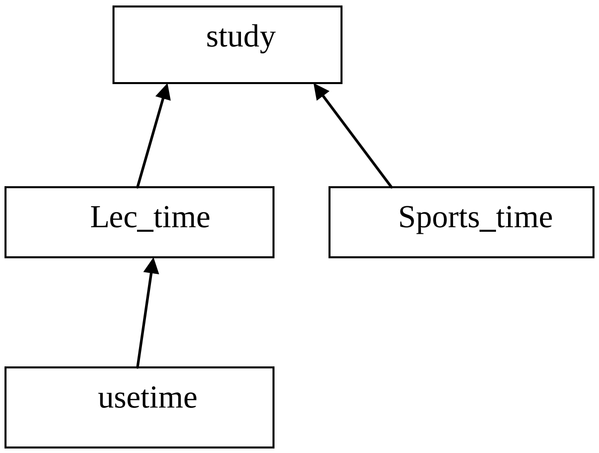


Figure 5.11: Inheritance Hierarchy of Case Study

Source Code of Considered Case Study

```
class study
{
public:
schedule(int a,int b)
5.{
6.get_lec_time(a,b);
7.if(hour<=3)
8.{
9.int ch=hour;
10.}
11.else
12.{
13.ch=4;
14.cout<<" college is closed";}
15.switch(ch)
16.case1:
17.cout<<"maths class";
18.break;
19.case2:
20.cout<<"physics class";
21.break;
22.case3:
23.cout<<"chemistry class";
24.break;
25.}
26.get_sports_time(a, b);
27.cout<<"the  sports time now is="a" hour "b" min";
28.}
29.}
class lec_time:public  study
{
int hours;
int minutes;
```

```
public:
void get_lec_time(int h,int m)
{
if (h<=12 && m<60)
{
hours=h;
minutes=m;
}
}
void put_lec_time(void)
{
cout<<hours<<"hours and";
cout<<minutes<<"minutes"<<\n;
}
}
class sports_time:public study
{
int hours;
int minutes;
void get_sports_time(int h,int m)
{
if(h<=12 && m<60)
{
hours=h;
minutes=m;
}
}
void put_sports_time(void)
{
cout<<hours<<”hours and”;
cout<<minutes<<”minutes”<<\n;
}
}
class usetime : public time
 {
 public:
 void sum (time t1, time t2)
 {
 minutes=t1.minutes+t2.minutes;
 hours=minutes/60;
 minutes=minutes%60;
 hours=hours+t1.hours+t2.hours;
 }
};
1.void main ()
2.{
3.lec_time t1, t2;
4.sports_time t4,t5;
```

```
5.usetime t3;
6.t1.get_lec_time (2, 45);               // get t1
7.t2.get_lec_time (3, 30);               // get t2
8.t3.sum (t1, t2);                       //t3=t1+t2
9.cout<<"t1="t1.put_lec_time();          //display t1
10.cout<<"t2="t2.put_lec_time();         //display t2
11.cout<<"t3="t3.put_lec_time();         //display t3
12.t1.schedule (5, 34);
13.t1.get_sports_time (2, 45);           // get t1
14.t2.get_sports_time (3, 30);           // get t2
15.t3.sum (t1, t2);                      //t3=t1+t2
16.cout<<"t1="t1.put_sports_time();      //display t1
17.cout<<"t2="t2.put_sports_time();      //display t2
18.cout<<"t3="t3.put_sports_time();      //display t3

19.getch();
20.}
```

First Level Prioritization (P, n)

The following calculations show the testing effort for each class and the priority assigned to each class.

At Level 1

Study class has highest priority.

Level=1

No of descendent classes=2

No of inherited attributes/methods=0

Testing effort=(2+0)*(4-1)=2*3= 6

At Level 2

For class *Lec_time*

Level=2

No of descendent classes=1

No of inherited attributes/methods=1

Testing effort=(1+1)*(4-2)= 2*2= 4

For class *Sports_time*

Level=2

No of descendent classes=0

No of inherited attributes/methods=1

Testing effort=(1+0)*(4-2)=1*2=2

Now,assign priorities on the basis of testing effort values.

Priority(*Lec_time*)> priority(*Sports_time*)

At Level 3

For class *Usetime*

Level=3

No of descendent classes=0

No of inherited attributes/methods=3

Testing effort=(3+0)*(4-3)=3*1=3

Priority(Study)> priority(Lec_time)> priority(Sports_time)> priority(Usetime)

Table 5.10: Priority Assigned to Each Class of Inheritance Hierarchy

Class name	**Priority number**
Study	1
Lec_time	2
Sports_time	3
Usetime	4

Lower number indicates higher priority.

Second Level Prioritization

Test Case Prioritization of Class Study

The test cases of each class in the inheritance heirarchy are to be prioritized on the basis of fault coverage using second level prioritization.

Flow graph of class study with labelled edges

The labelled flow graph of class study is shown in Figure 5.12 The independent paths have been recognized using flow graph and the test cases has been designed using independent paths.

Test Cases of Class STUDY

On the bases of independent path test cases are designed. Test cases are shown in Table 5.11.

Table 5.11: Test Cases of Class Study

Test Case	Path Covered
TC1	ABCDEFHJKNQRSTU
TC2	ABCDEGIJKNQRSTU
TC3	ABCDEFHJLOQRSTU
TC4	ABCDEGIJLOQRSTU
TC5	ABCDEFHJMPQRSTU
TC6	ABCDEGIJMPQRSTU

Fault can be detected in class STUDY:

Fault1:- at node D, in definition of function

Fault2:- At E node, checking condition

Fault3:-at node J, switch statement

Fault4:-at node K

Fault5:-at node L

Fault6:-at node M

Fault7:-at node R

Each fault is assigned a weight using Table 5.9.

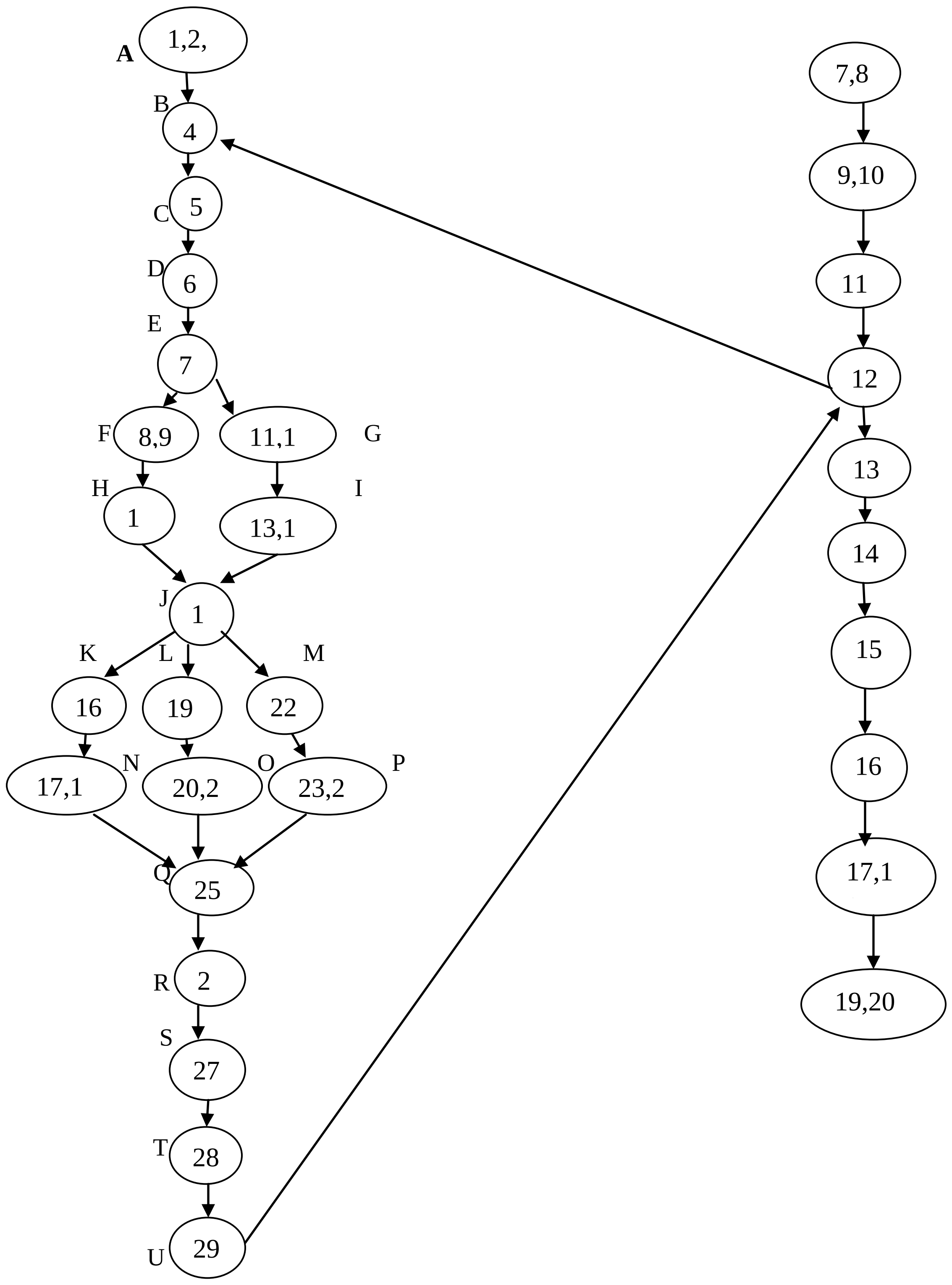


Figure 5.12: Flow Graph of Class Study

The faults of class study are assigned weight as shown in Table 5.12.

Table 5.12: Faults Weight (Class Study)

Fault Number	**Fault Weight**
1	2
2	2
3	2
4	1
5	1
6	1
7	1

Test cases and faults of class STUDY are shown in Table 5.13.

Table 5.13: Faults Detected by Non Prioritized Test Cases (Class Study)

	TC1	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**
F1(2)	*	*	*	*	*	*
F2(2)	*	*	*	*	*	*
F3(2)	*	*	*	*	*	*
F4(1)	*	*				
F5(1)			*	*		
F6(1)					*	*
F7(1)	*	*	*	*	*	*
total fault	8	8	8	8	8	8
time taken	5	7	11	4	10	12

APFD Result of Test Suite before Prioritization:-

Where TFi=ith fault is detected by which test case.

n=total number of test cases

m=total number of fault

$$APFD=1-\frac{TF1+TF2+\ldots\ldots\ldots\ldots+TFm}{n*m}+\frac{1}{2n}$$

TF1=1 TF2=1

TF3=1	TF4=1
TF5=3	TF6=5
TF7=1	

$$APFD = 1 - \frac{(1+1+1+1+3+5+1)}{6*7} + \frac{1}{2*10}$$

APFD= 78%

Prioritization of test suite based on proposed algorithm

Step1: RFD = fault/time (rate of fault detection)

RFDTC1=8/5=1.60	RFDTC2=8/7=1.14
RFDTC3=8/11=0.72	RFDTC4=8/4=2.0
RFDTC5=8/10=0.80	RFDTC6=8/12=0.66

Step2: Sorting of RFD

TC4, TC1, TC2, TC5, TC3, TC6

Step3: Remove TC4 from T and add TC4 to T'

Now T' ={ TC4}

T = {TC1, TC2, TC3, TC5, TC6}

Step 4: Until T Is Not Empty

Step5: New Fault Coverage of Test Cases Per Unit Time

RFDTC1=1/5=0.20	RFDTC2=1/7=0.14
RFDTC3=0	RFDTC5=1/10=0.10
RFDTC6=1/12=0.08	

STep6: Remove TC1 from T and Add To T'

T = {TC2, TC3, TC5, TC6}

T' ={TC4, TC1}

Step7: Go to Step 4

Step4: Until T Is Not Empty

Step5: New Fault Coverage of Test Cases Per Unit Time

RFDTC2=0.00	RFDTC3=0
RFDTC5=1/10=0.10	RFDTC6=1/12=0.08

Step6: Remove TC5 from T and Add To T'

T = {TC2, TC3, TC6}

T'={TC4, TC1, TC5}

Step7: Go to Step 4

Step4: Until T is Not Empty

Step5: New Fault Coverage of Test Cases Per Unit Time

RFDTC2=0.00	RFDTC3=0.00
RFDTC6=1/12=0.00	

Step6: Remove TC7 from T and Add To T'

T' ={TC4, TC1, TC5, TC2,TC6,TC3}

Step8: Return T' which is Prioritized Test Suite.

Prioritized test suite is shown in Table 5.14.

Table 5.14: Faults Detected by Prioritized Test Cases (Class Study)

	TC1(TC4)	TC2(TC1)	TC3(TC5)	TC4(TC2)	TC5(TC6)	TC6(TC3)
Fault 1	*	*	*	*	*	*
Fault 2	*	*	*	*	*	*
Fault 3	*	*	*	*	*	*
Fault 4		*		*		
Fault 5	*					*
Fault 6			*		*	
Fault 7	*	*	*	*	*	*

APFD of Prioritized Test Suite is 85%

Fault percent detection corresponding to each test case of random and prioritized test suites are shown in Table 5.15.

Table 5.15: Percentage of Faults Detected by Test Cases

Non Prioritized Test Cases	**Fault % detected**	**Prioritized Test Cases**	**Fault % detected**
TC1	71.4	TC1	71.4
TC2	71.4	TC2	85.7
TC3	85.7	TC3	100
TC4	85.7	TC4	100
TC5	100	TC5	100
TC6	100	TC6	100

Comparison of Prioritized and Non Prioritized Test Suite

Based on the analysis done in previous section the prioritized test suite is better as compare to non prioritized test suite. Using APFD metric comparison is shown in Table 5.16.

Table 5.16: APFD Metric (Class Study)

TEST CASES	APFD %
Non Prioritized	78%
Prioritized	85%

Fault percent detected by test case of non prioritized and prioritized test suite is shown in Figure 5.13 and Figure 5.14 respectively.

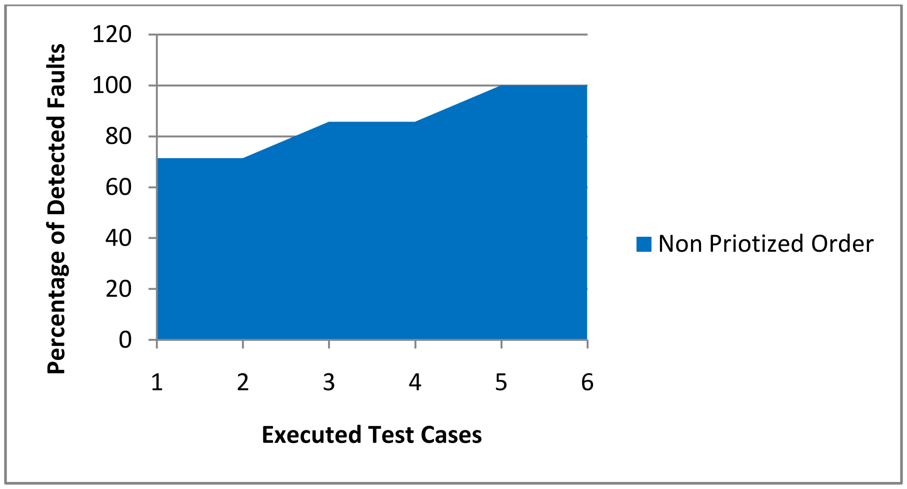


Figure 5.13: Percentage of Faults Detected by Non Prioritized Test Suite

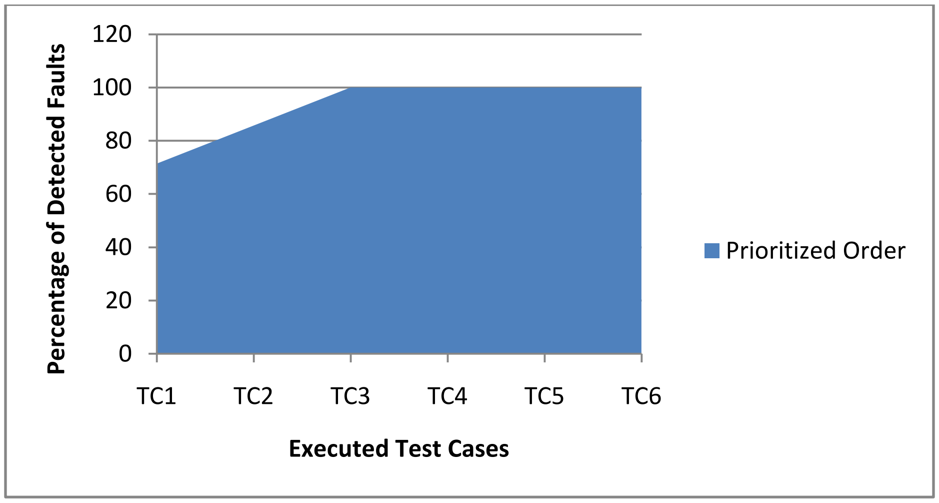


Figure 5.14: Percentage of Faults Detected by Prioritized Test Suite

Test Case Prioritization of Class lec_time

```
1.class lec_time:public  study
2.{
3.int hours;
4.int minutes;
5.public:
6.void get_lec_time(int h,int m)
7.{
8.if (h<=12 && m<60)
9.{
10.hours=h;
11.minutes=m;
12.}
13.}
14.void put_lec_time(void)
15.{
16.cout<<hours<<"hours and";
17.cout<<minutes<<"minutes"<<\n;
18.}}
```

Flow Graph of Class lec_time

Flow graph of class lec_time is shown in Figure 5.15.

Test Cases of Class lec_time: Test cases of class lec_time are shown in Table 5.17.

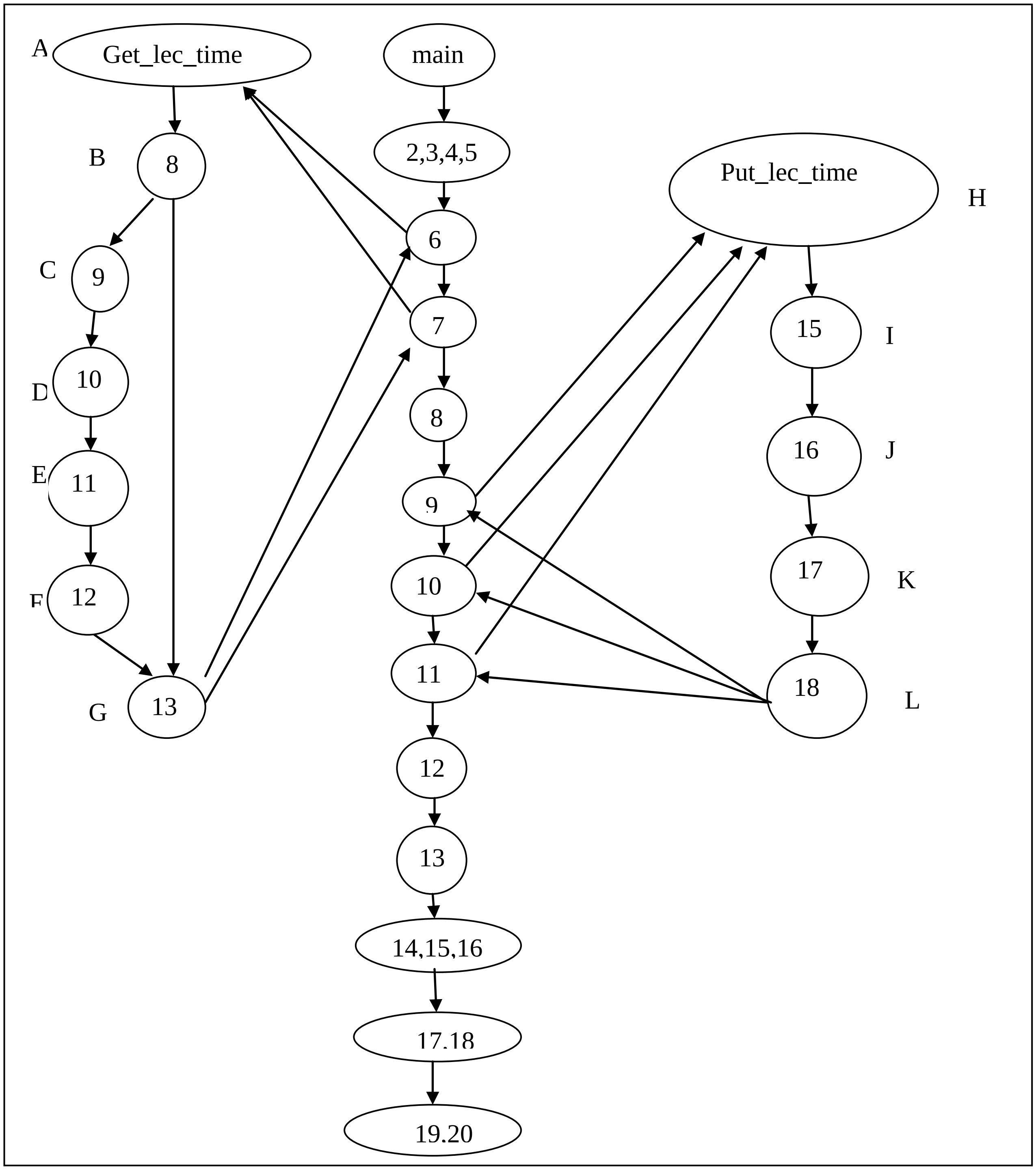


Figure 5.15: Flow Graph of Class Lec_time

Table 5.17: Test Cases of Class lec_time

Test Case	**Path Covered**
TC1	ABCDEFG
TC2	ABG
TC3	HIJKL

Faults of class lec_time are shown below:-

Fault 1:-Type mismatch of arguments at node A

Fault2:-check condition at node B

Fault3:-statement in if block

Fault4:-type mismatch of arguments at node H

The faults are assigned weight using Table 5.9 based on the structure of flow graph as shown in Table 5.18.

Table 5.18: Faults Weight (Class Lec_time)

Fault Number	**Fault Weight**
1	2
2	2
3	1
4	2

The test case and fault table of class lec_time are shown in Table 5.19

Table 5.19: Test Case and Detected Faults of Class lec_time

Fault Name & Weight	TC1	TC 2	TC3
Fault 1 (2)	*	*	
Fault 2 (2)	*	*	
Fault 3 (1)	*		
Fault 4 (2)			*
Total fault weight	5	4	2
Time taken	6	2	3

Prioritized Order of Test Cases of Class Lec_time

The test suite is prioritized on the basis fault detection per unit time of test cases:

TC1, TC3, TC2

Because fault detection per unit time of test case 2 is more than that of test case 1and 3

Comparison of Prioritized and Non Prioritized Test Suite

Based on the analysis done in previous section the prioritized test suite is better as compare to random test suite. Using APFD metric comparison is shown in Table 5.20.

Table 5.20: APFD Metric (Class Lec_time)

TEST CASES	**APFD %**
Non Prioritized	66.67 %
Prioritized	75 %

Fault percent detection corresponding to each test case of non prioritized and prioritized test suites are shown in Table 5.21:

Table 5.21: Percentage of Faults Detected by Test Cases

Non Prioritized Test Cases	Fault % detected	Prioritized Test Cases	Fault % detected
TC1	75	TC1	75
TC2	75	TC2	100
TC3	100	TC3	100

Fault percent detected by test case of non prioritized and prioritized test suite is shown in Figure 5.16 and Figure 5.17 respectively.

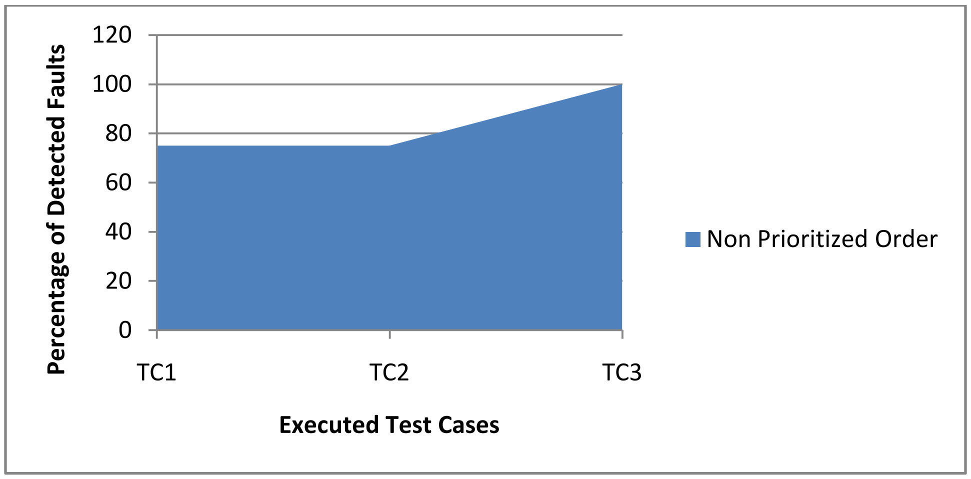


Figure 5.16: Percentage of Faults Detected by Non Prioritized Test Cases

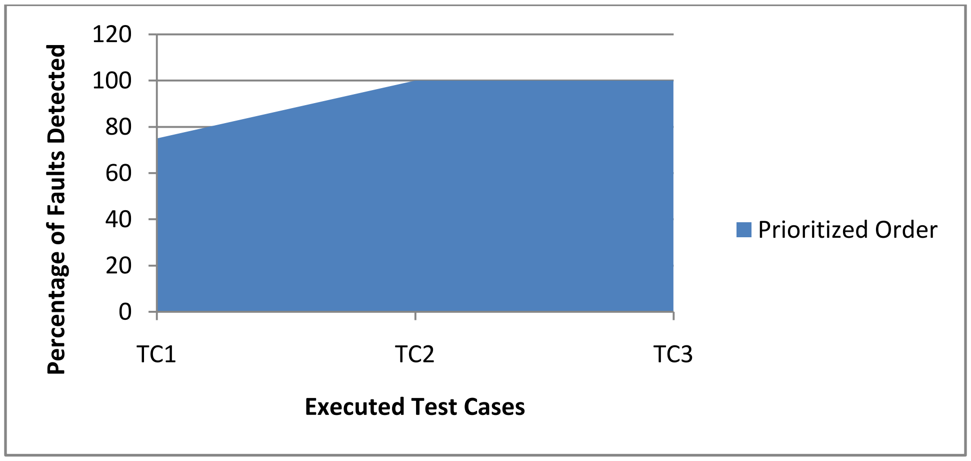


Figure 5.17: Percentage of Faults Detected by Prioritized Test Cases

Test Case Prioritization of Class sports_time

```
1.class sports_time:public study
2.{ 3.int hours;
4.int minutes;
5.public:
6.void get_sports_time(int h,int m)
7.{
8.if(h<=12 && m<60)
9.{
10.hours=h;
11.minutes=m;
12.}
13.}
14.void put_sports_time(void)
15.{
16.cout<<hours<<”hours and”;
17.cout<<minutes<<”minutes”<<\n;
18.}
19.}
```

Flow Graph of Class sports_time: The flow graph for class sports_time is shown in Figure 5.18.

Test Cases of Class sports_time : The test case of class sports_time are shown in Table5.22:

Table 5.22: Test Cases of Class sports_time

Test Case	Path Covered
TC1	ABCDEFG
TC2	ABG
TC3	HIJKL

The faults detected for class sports_time are shown below:

Fault 1:-Type mismatch of arguments at node A

Fault2:-check condition at node B

Fault3:-statement in if block

Fault4:-type mismatch of arguments at node H

The Faults are assigned weight using Table 5.9 based on structure of flow graph as shown in Table 5.23

Table 5.23: Faults Weight (sports_time)

Fault Number	Fault Weight
1	2
2	2
3	1
4	2

The test cases and faults of class sports_time are shown in Table 5.24

Table 5.24 Test Case and Detected Faults of Class sports_time

Fault Name & Weight	TC1	TC2	TC 3
Fault 1 (2)	*	*	
Fault 2 (2)	*	*	
Fault 3 (1)	*		
Fault 4 (2)			*
Total fault weight	5	4	2
Time taken	6	2	3

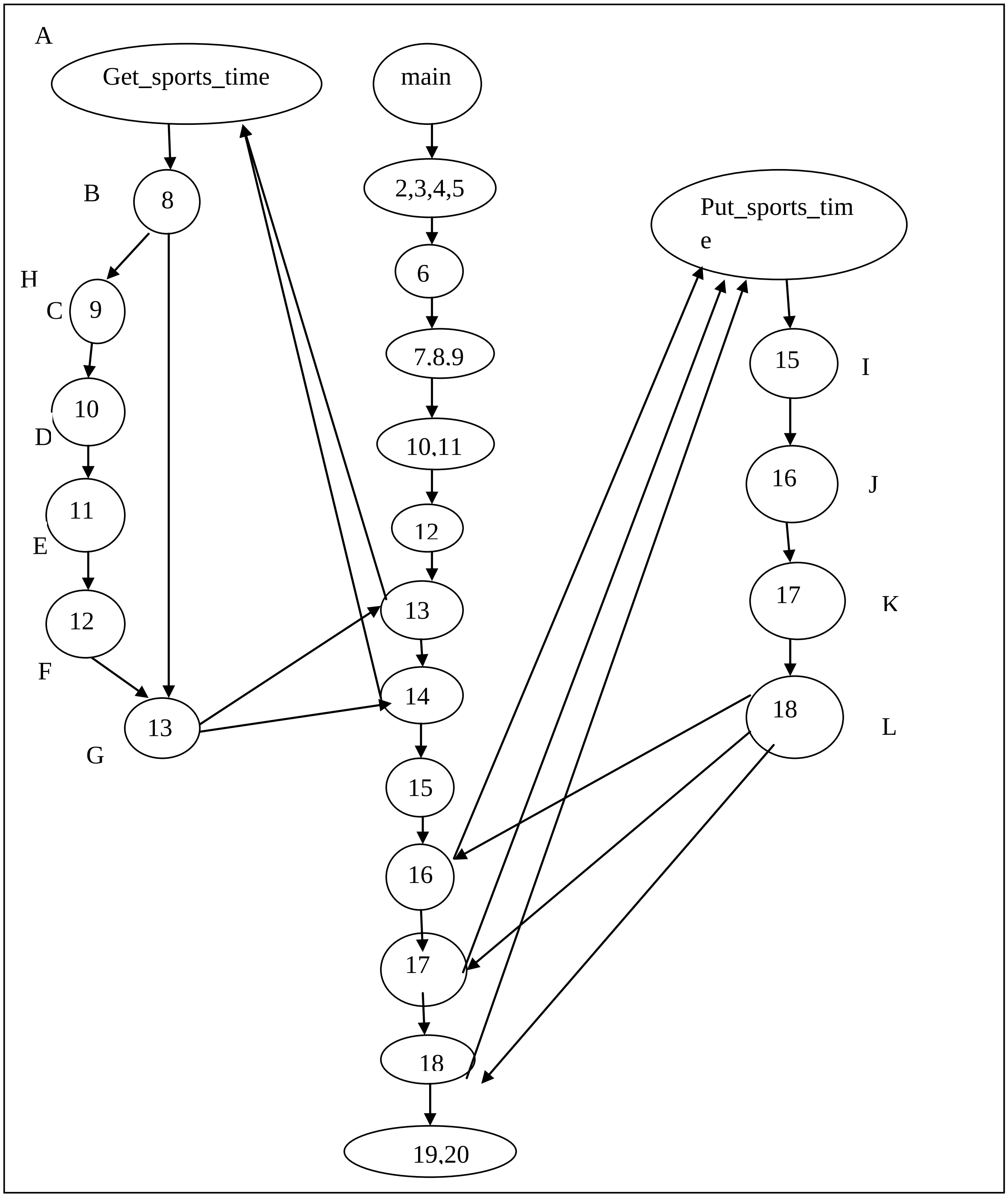


Figure 5.18: Flow Graph of Class sports_time

Prioritized Order of Test Case of Class sports_time

The test suite is prioritized on the basis of fault detection per unit time of test cases:

TC1, TC3, TC2

Because fault detection per unit time of test case1 is more than that of test case 2 and test case 3.

Therefore TC1 is ordered first, TC2 is ordered second and TC3 is ordered third.

Comparison of Prioritized and Non Prioritized Test Suite

Based on the analysis done in previous section the prioritized test suite is better as compare to random test suite. Using APFD metric comparison is shown in Table 5.25.

Table 5.25: APFD Metric (Class sports_time)

Test CASES	**APFD %**
Non Prioritized	66.67 %
Prioritized	75 %

Fault percent detection corresponding to each test case of non prioritized and prioritized test suites are shown in Table 5.26.

Table 5.26: Percentage of Faults Detected by Test Cases

Non Prioritized Test Cases	**Fault % detected**	**Prioritized Test Cases**	**Fault % detected**
TC1	75	TC1	75
TC2	75	TC2	100
TC3	100	TC3	100

Fault percent detected by test case of non prioritized and prioritized test suite is shown in Figure 5.19 and Figure 5.20 respectively.

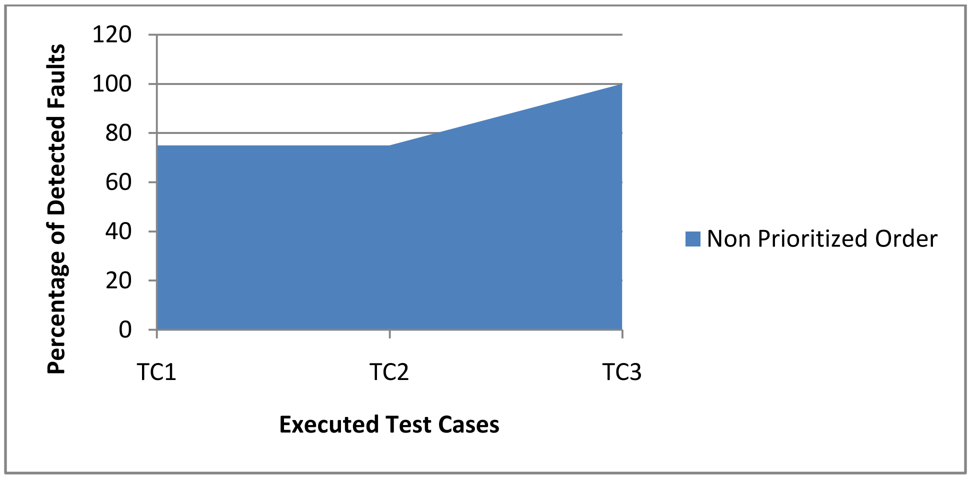


Figure 5.19: Percentage of Faults Detected Non Prioritized Test Cases

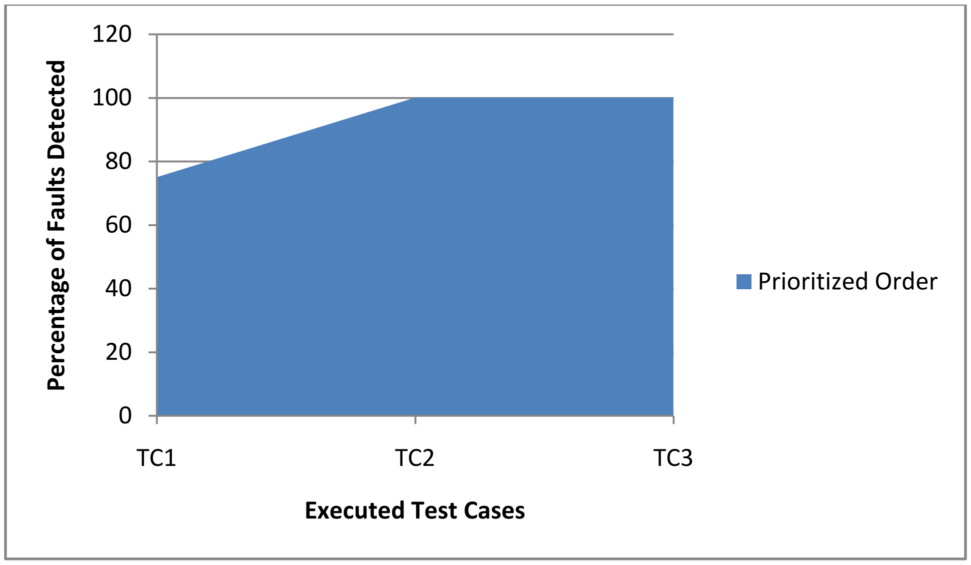


Figure 5.20: Percentage of Faults Detected by Prioritized Test Cases

Based on the analysis done in previous section the prioritized test suite is better as compare to non prioritized test suite.

This is clear from the area under curve that fault percent detection by prioritized test suite is better as compared to random test suite.

Test Case Prioritization of Class usetime

```
class usetime : public time
{
public:
void sum (time t1, time t2)
{/
minutes=t1.minutes+t2.minutes;
hours=minutes/60;
minutes=minutes%60;
hours=hours+t1.hours+t2.hours;
}
};
```

On the basis of code coverage test cases are designed here. There is only one function in class 3.Only one test case is enough to cover all statements of function sum in class 3. So there is no need to prioritize test cases of class 3.

The proposed technique is implemented on a case study and results are analyzed by average percentage of fault detection metric. The result of proposed technique is shown in Table 5.27.

Table 5.27: Result of Proposed Technique

Class study	TC4	TC1	TC5	TC2	TC6	TC3
Class lec_time	TC2	TC1	TC3			
Class sports_time	TC2	TC1	TC3			
Class usetime	TC1					

The comparison using APFD metric is shown in Table 5.28.

Table 5.28: Analysis of APFD Metric

Class name	**Non Prioritized Test Cases**	**Prioritized Test Cases**
Study	78	85
Lec_time	66.67	75
Sports_time	66.7	75

The analysis shows that proposed technique is better as compared to non prioritized test case prioritization approach.

5.4 A HISTORY BASED TECHNIQUE FOR REGRESSION TEST CASE PRIORITIZATION OF OBJECT ORIENTED SOFTWARE (HTRTCPOOS)

The proposed approach prioritizes the regression test cases on the basis of some factors related to the past testing history and coverage of the code in term of classes of the software which is going to be retested after incorporating some modifications in it. All the considered factors have been shown in the Table 5.29. All the factors have been assigned a positive weight which shows the capability of the test cases to discover the maximum fault by consuming less time and cost. The weights of factors are totally probabilistic. To assign the weight of factors a survey has been performed (See Appendix F). The Participants participated in the survey are the Developer, Tester, Lead Technology, etc. These factors may be considered for the prioritization factor for the regression testing of the software. The value of the considered factors is determined by using the information of past history of the test cases.

Table 5.29: Prioritization Key of Test Cases (HTRTCPOOS)

S.No.	**Factor Name**	**Factor Weight**
1	Severity of Bug	.25
2	Capability of Detecting the Bug	.2
3	Coverage of code	.15
4	Impact on business	.3
5	Execution Time	.1

The test cases are thus prioritized on the basis of a value known as regression test case prioritization value (RTCPV) which is calculated by the following Formula

$$RTCPV = \sum_{i=1}^{n} TFV_i * FW_I \quad \text{--}(5.1)$$

Where TFV is the estimated value of the i_{th} factor and FW is factor weight of i_{th} factor of test case.

In regression test cases if the test cases are new then it is assigned the highest priority because it is going to be executed first time and has the capability of detecting the maximum faults. It may be possible that new test cases are more than one. In such type of dilemma the newly test cases are prioritized on the basis of coverage of modified classes and coverage of new classes

In the presented approach all the detected bugs are classified in different category on the basis of the severity of the bugs. The five factors have been considered for prioritizing the test cases. Every factor has been assigned a positive weight and value will be calculated on the basis of the past history of the test cases. The overall process of test case prioritization is shown in Figure 5.21, which is being a described further in subsequent sections.

5.4.1 The Prioritization Factors Considered in the Presented Approach

(a) Severity of Bug: This factor use the classification of the bug on the basis of the their impact on the software. Here the bugs are classified in the four categories. These categories are critical bug, major bug, medium bug and minor bug. Here on the basis of the past discovery of the bugs by test cases a scaling of bugs (0-10) may be given as below in Table 5.30.

(b) Capability of Detecting the Bug (CDB): This [108] factor shows the caliber of the test case to detect the maximum bugs by executing the test cases. The value of this can be estimated by the Formula 5.2.

CDB = (TBC/TDB) *10 ---(5.2)

Where TDB is the total detected bug by all test cases and TBC is number of bugs detected by the current test cases

(c) Coverage of Code (CC): This factor shows the coverage of the code in terms of classes (modified and unmodified) and methods by the test cases. The value of this factor is based on the basis of coverage of the modified and updated classes. This value can be calculated by the following Formula

CC = (TCC / TC)*10-------------------------------(5.3)

Where TC is Total classes in the software and TCC is number of covered classes by the test cases. On the basis of this formula the value between 0 to 10 is assigned

(d) Business Impact: This factor shows that if the particular function being covered by the test cases is not executed successfully then how much it puts impact on the business of customer. On the basis of the business impact by test cases the value between 0 to 10 is assigned.

(e) Execution Time (ET): This factor shows the time taken by the test case to execute the target functionality. The value of this factor is assigned on the basis of the formula given below

ET = (PT/TT)*10 ------------------------------------(5.4)

Where PT is execution time i_{th} test case , TT is the total time taken in executing all test cases and ET is the estimated value of execution time of the particular test cases.

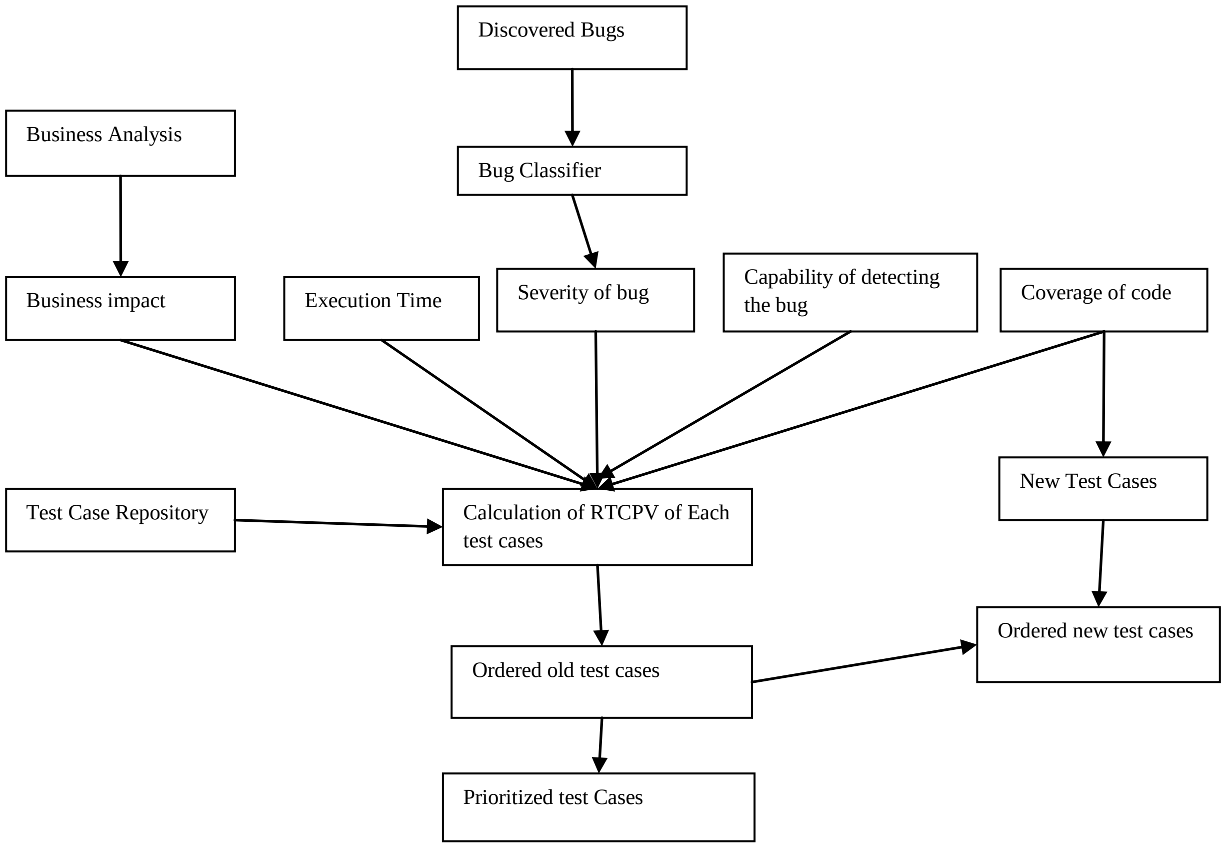


Figure 5.21: Overview of Proposed Approach (HTRTCPOOS)

Table: 5.30: Value Assigned to the Detected Faults

S.No.	**Value**	**Categories of Bugs**
1	10	All critical Bugs
2	8-9	Critical, major, medium and minor bugs
3	7	All Major Bugs
4	5-6	Major and Medium
5	4	All Medium Bugs
6	2-3	Medium and Minor
7	1	All Minor Bugs

5.4.2 Result and Analysis:

For the experimental applicability and analysis of the proposed approach, it has been applied on a case study [169] implemented in Java. To check effectiveness of the technique to detect rate of fault detection, intentionally some faults have been added in the considered case study and the bugs are detected manually.

5.4.3 Case Study: In this case study the presented approach is applied on a practical problem of Banking. In the considered example [169] the user can perform the operation of deposit, withdrawal, calculate interest, and display the account information on saving account and current accounts. Table 5.31 shows the test case history of the program before applying the modification

Table 5.31: Testing History of Consider Case Study

Test case	**Count of detected Bugs**	**Severity of Bug**	**Execution time of test case (cs)**
TC1	1	Minor=1	.2
TC2	2	Major =1 , minor=1	.3
TC3	1	major=1	.25
TC4	1	Minor=1	.2
TC5	1	Major =1	.25
TC6	2	Minor=2	.25
TC7	2	Major=2	.3
TC8	2	Major=1 ,Medium=1	.35
TC9	3	Critical =1, Major =2	.35
TC10	1	Medium=1	.2

From the past testing history of the case study the total 16 bugs are discovered by executing the 10 test cases. Now by using the above history the Table 5.32 shows the values of various factors which are used to prioritize the test cases for regression testing.

Table 5.32: Determined Value of Considered Factors

Test case	**Determined value of severity of Bug**	**Capability of Detecting the bug (CDB)**	**Execution time of test case (ET)**	**Impact on business**	**Coverage of code (CC)**	**Estimated RTCPV**
TC1	1	(1/16)*10= .625	(.2/2.65)*10 =0.75	2	(4/5)*10 =8	(1*.25) +(0..625*.2)+(.75*.1)+(2*.3)+(8*.15) = 2.25
TC2	7	.80	1.13	8	8	5.623
TC3	5	.625	.94	8	8	5.069
TC4	1	.625	.75	9	8	4.35
TC5	5	.625	.94	5	8	4.169
TC6	1	.80	.94	2	8	2.304
TC7	5	.80	1.13	8	8	5.122
TC8	7	.80	1.32	9	8	5.942
TC9	9	1.87	1.32	9	8	6.656
TC10	3	.625	0.75	7	8	4.249

The ordered test cases are TC9, TC8, TC2, TC7,TC3, TC4, TC10,TC5,TC6,TC1

The Table 5.33 shows the order of the test cases after applying the random, reverse, Nayak et al. [108] and the proposed approach

Table 5.33.Test Case Order of the Various Approaches and Proposed Approach

S.No.	**Non Prioritized**	**Nayak approach**	**Proposed approach**
1	TC1	TC9	TC9
2	TC2	TC2	TC8
3	TC3	TC7	TC2
4	TC4	TC8	TC7
5	TC5	TC6	TC3
6	TC6	TC5	TC4
7	TC7	TC3	TC10
8	TC8	TC10	TC5
9	TC9	TC1	TC6
10	TC10	TC4	TC1

The Table 5.34 shows the faults detected by the test cases.

Table 5.34: Faults Detected by Test Cases

	TC1	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**	**TC7**	**TC8**	**TC9**	**TC10**
F1	*					*				
F2								*		
F3		*								
F4		*								
F5			*	*						
F6			*							
F7			*							
F8				*						
F9					*					
F10							*			
F11								*		
F12									*	
F13									*	
F14									*	
F15										*

The APFD of non prioritized, Nayak approach and the proposed approach is shown in Figure 5.22 to Figure 5.24

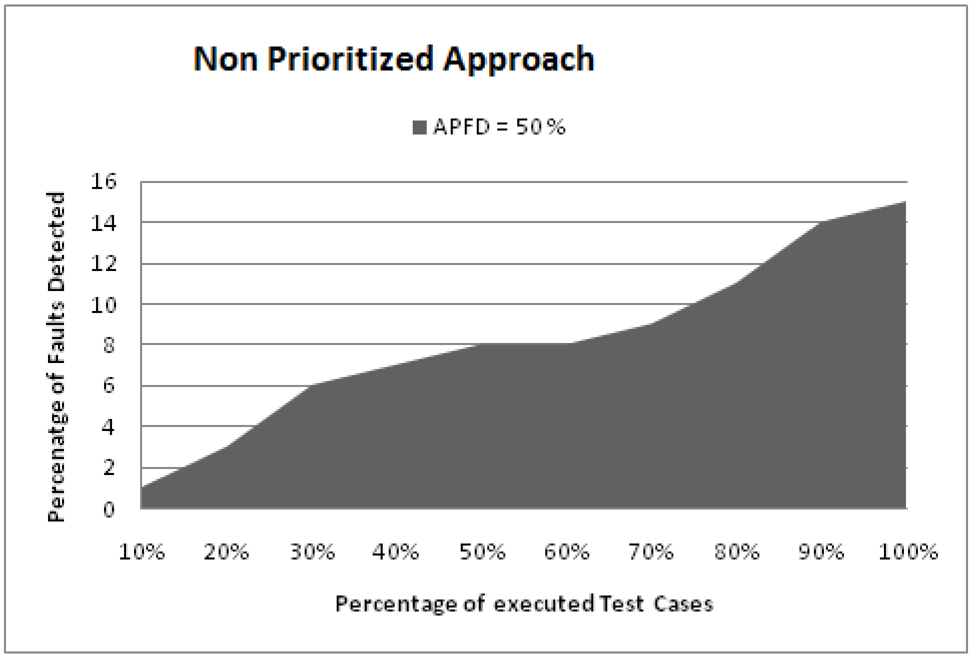


Figure 5.22: APFD Graph of the Unordered Test Cases

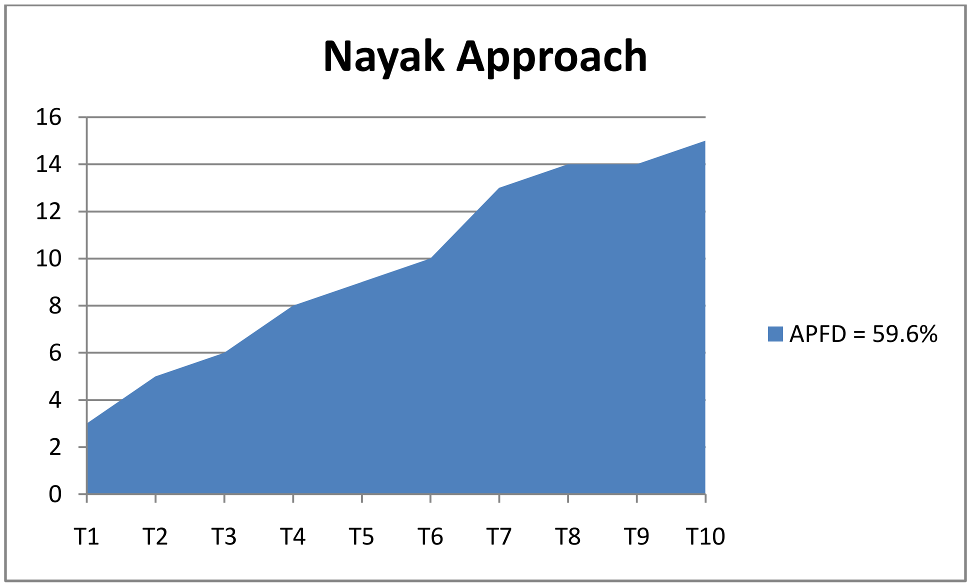


Figure 5.23: APFD Graph of the Test Cases Ordered by Nayak et. al. Approach

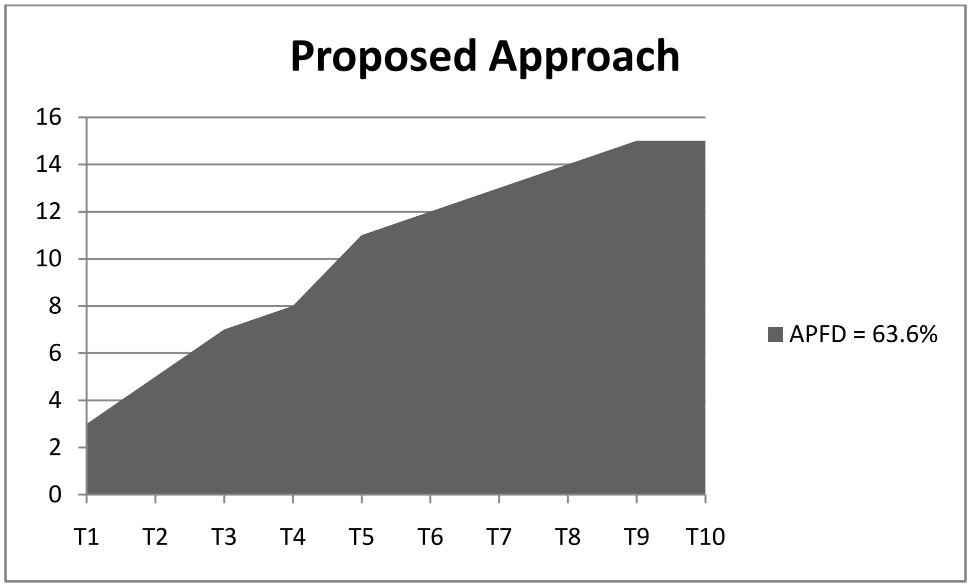


Figure 5.24: APFD Graph of the Test Cases Ordered by Proposed Approach

5.4.4 Comparative Study of the Proposed Approach

The Figure 5.25 and Table 5.35 show that the proposed approach is to discover the maximum faults earlier as compare to the other approaches. The result of the proposed approach is very promising and helps to reduce the testing cost of the software.

Table 5.35: APFD Value of the Proposed Approach and Others Approaches

S.No.	**Approach Applied**	**Percentage of APFD**
1	Non Prioritized Approach	50%
2	Nayak et. al. Approach	59.6%
3	Proposed Approach	63.6%

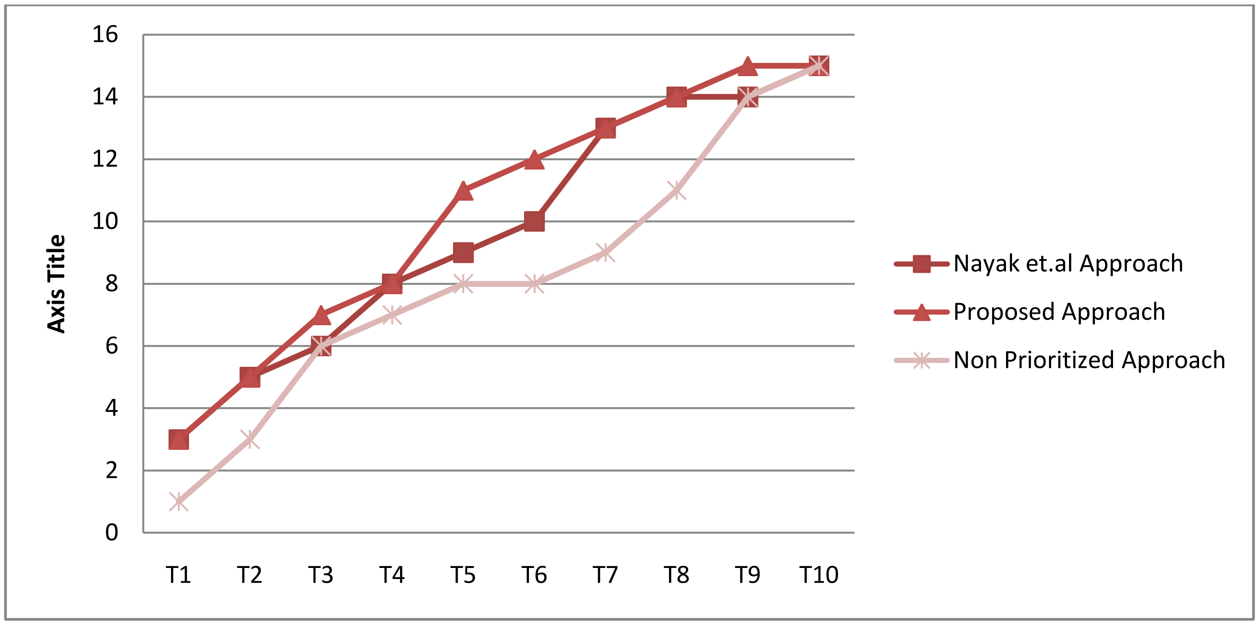


Figure 5.25: Comparison of APFD Graph of Various Approaches

5.5 CONCLUSION

In this chapter test case prioritization techniques to prioritize the regression test suite have been presented. In the first technique test cases are selected using the OPDG and dynamic slicing. In the second, classes are prioritized on the basis of the calculated testing effort followed by the prioritization of test cases of the prioritized class based on the types of fault detected by the test cases. In third technique, some past history factors have been considered to prioritize the test cases. Every factor has been assigned a positive weight which helps to detect the maximum faults. For experimental validation and applicability all the techniques have been applied on the different software. The result shows the efficacy of the techniques

Chapter VI

CONCLUSION AND FUTURE SCOPE

6.1 CONCLUSION

In the proposed research work various techniques to prioritize the test cases for Unit, Integration testing, System testing and Regression testing for Object Oriented Software has been presented. To prioritize the test cases various factors have been considered. In some techniques various surveys have been performed to determine the capability of the factors to detect the faults. To prioritize the test cases of System testing, a Data Mining tool named SPSS modeler is used on the four algorithms to determine the proposed factors weight. For experimental verification and validation the proposed techniques have been applied on the many software implemented in C++ and Java. To analyze the effectiveness of the proposed approach the experimented results are compared with other exited similar techniques and random approaches. From the outcomes it has been observed that the proposed techniques help to reduce the testing cost and time to test the software.

6.2 BENEFITS OF THE PROPOSED WORK

- **Determination of the Affected Part of the Modified Software**

  With the help of the presented approach the tester can determine the affected part of software by introducing the changes in the software. The tester can easily determine the all the affected paths and select the test cases corresponding to the determined affected paths. This helps to reduce the number of test cases which are needed to be execute to ensure the correctness of the software.

- **Reduction of Testing Cost and Time**

The objective of the test case prioritization techniques to execute the test cases to determine the maximum faults as earlier stage by utilizing the minimum resources and time. It is very costly to detect and fix the bug at later stages. The detection of the maximum faults as earlier stages of the software life cycle helps to reduce the testing cost, time.

- **Improve the Quality and Reliability of the Software:**

In the presented work many test case prioritization technique consider the various factors to prioritize the test cases. These factors are chosen on the basis of their capability to introduce the errors in the software or if their impact on the working of the software if they are not implemented in proper way. So by applying these techniques they enhance the quality of the software.

- **Customer Satisfaction:**

With the help of presented work the quality software delivered to the customer within the specified time without any delay. The developed software is able to fulfill all the functionality as desired by the customer.

6.3 FUTURE SCOPE

The work contained in this research work can be extended with the following list of possible future research issues in Object oriented software.

- **Test Case Prioritization of Object Oriented Technique using Data Mining Algorithms**

  The algorithm used for mining the data is very helpful to retrieve the useful information. Data mining technique may be used to prioritize the data by analyzing the past history data of testing from the industry. Various algorithms may be used to identify the relation present between the faulty test cases which further helps to detect the maximum faults earlier as possible

- **Testing the Proposed Techniques for the Industry Projects**

  The proposed test case prioritization techniques have been tested on small projects. It would be better if these are applied on large scale industry projects.

- **Test case prioritization for Acceptance level testing**

  In this thesis the test case are prioritized for unit, Integration, system and regression testing levels. The acceptance testing is very challenging process which may affect the quality of the software. It would be better to analyze the factors that must be considered to prioritize the test cases of acceptance testing.

- **Test Case prioritization by Identifying the Human Factors**

  Various new technologies have been used to reduce the testing cost of the software. It will increase day by day. So human factors play a very important role to produce quality software. It will be beneficial to identify and analyze the human factors such as stress, motivation etc. to prioritize the test cases at unit, integration, system testing

REFERENCES

[1] G. Rothermel, R.H. Untch, Chengyun Chu, M.J. Harrold "Prioritizing test cases for regression testing" *IEEE Transactions on Software Engineering* Volume: 27, Issue 10, Oct 2001 Page(s) 929 - 948

[2] Naresh Chauhan *Software Testing Principles and Practices*. New Delhi Oxford University Press, 2010

[3] Imran Bashir, Amrit L. Goel *Testing Object Oriented Software Life Cycle Solutions*. Verlag Springer New York ,1999

[4] N. Hunt. "Performance Testing C++ code" *Journal of Object Oriented Programming,* Jan 1996, pages 22-25,.

[5] E. V. Berard . "Essays on Object Oriented Software Engineering" Volume1. Prentice Hall, 1992

[6] G. Booch. "Object Oriented Analysis and Design with Applications" Benjamin /cummings, 2nd edition, 1994

[7] D. DeChampeaux, D. Lea, and P. faura. "Object – Oriented Systems Development" . *Addison – Wesley*, 1993

[8] N. Hunt. "Unit Testing" *Journal of Object Oriented Programming* pages 18-23, Feb, 1996.

[9] J. O. Coplien. "Advanced C++ Programming Styles and Idioms" *Addison Wesley* 1992

[10] B. Hetzel. "The Complete Guide of Software Testing". *John Wiley & Sons*, 2nd edition , 1988

[11] Yogesh Singh, *Software Testing*. Delhi Combridge University Press, 2012.

[12] Object Oriented Design Principle http://www.oodesign.com/design-principles.html

[13] W stevens . G. Myers and L. Constantine " Structured Design" *IBM System Journal*. Vol. 13 , 1974, PP. 115 -130 ,.

[14] Johann Eder, Gerti Kappel and Michael Schrefl "Coupling and Cohesion in Object Oriented Systems"

https://pdfs.semanticscholar.org/48ec/8e707053100b253cdee1f4aa56399a7ce94c.pdf

[15] Durga Prasad Mohapatra, Rajib Mall and Rajeev Kumar "An Overview of Slicing Techniques for Object-Oriented Programs" *Informatica* 30 ,2006 253–277

[16] M. J. Harold, J. A. Jone, T. Li, and D. Liang, "Regression test selection for java software," in *Proc. of the ACM Conference on OO Programming, Systems, Languages, and Applications*, 2001.

[17] E. Engstrm and P. Runeson, "A Qualitative Survey of Regression Testing Practices," *in Proc. of the International Conference on Product-Focused Software Process Improvement*, 2010, pp. 3–16.

[18] M. J. Harrold, J. D. McGregor, and K. J. Fitz- patrick, "Incremental Testing of Object Oriented Class Structure", *Proc. of 14th International Conf. on Software Engineering*, 1992, pp. 68 - 80.

[19] G. Rothermel and M. J. Harrold. "Analyzing Regression Test Selection Techniques" *IEEE Transactions on Software Engineering*, 22(8) 1996, 529-551.

[20] S. Elbum, A. Malishevsky, G Rothermel " Prioritization Test Cases For Regression Testing" Proceeding of International symposium on Software Testing and Analysis, 2000, PP 102-112.

[21] Gupta V , Chhabra JK, "Package Coupling Measurement In Object Oriented Software" Journal of Computer science& Technology 24(2), 2009, 273-283 Mar. .

[22] Vipin Sexena and Santosh Kumar, "Impact of Coupling and Cohesion In Object Oriented Technology" *Journal of Software Engineering and Application* , 2012 ,5, 671- 676

[23] Roger T Alexander and Jeff Offutt, "Coupling based testing of O – O Programs" *Journal of Universal Computer Science* ,2004.

[24] Eric Arisholm Lionel C. Briand and Audun Foyen , "Dynamic Coupling Measurement for Object Oriented Software" *Simula TR and Carleton TR SCE* ,2003 ,03 -18.

[25] Varun Gupta and Jitender kumar Chhabra "Dynamic Cohesion Measures for

Object Oriented Software" *Journal of system Architecture* 57. 2011, 452–462p.

[26] Jeff Offut ,Avnur Abdruazic , Stephen R. Schach "Quantitatively Measuring Object Oriented Couplings" Online at http://www.uml.org/

[27] V. S. Bidve and Akhil Khare " Simplified Coupling Metrics for Object Oriented Software" *International Journal of Computer Science and Information Technology,* Vol. 3(2), 2012 , 3389 – 3842

[28] Roger T Alexander and Jeff Offut " Coupling – based Testing of O-O Programs" https://cs.gmu.edu/~offutt/rsrch/papers/ootest-jucs.pdf

[29] Roger T. Alexander , jeff offut and James M. Bieman " Fault Detection Capabilities of Coupling – based OO Testing" *Proceeding of International Conference Software Reliability Engineering (ISSRE 2002)* PP 207-218.

[30] Kailash Patidar, Ravinder kumar gupta and Gajendra singh Chandel "Coupling and Cohesion Measures in Object Oriented Programming" *International Journal of Advanced Research in Computer Science and Software Engineering,* Volume 3, Issue 3 March, 2013.

[31] Zhenvi Jin and Jefferson Offutt "Coupling based Criteria for Integration Testing" *Software Testing , Verification and Reliability* 8, 1998,133-154.

[32] Shahzada Zeeshan Waheed, Usman Qamar "Data Flow Based Test Case Generation Algorithm for Object Oriented Integration Testing" *2015 6th IEEE International Conference on Software Engineering and Service Science* (ICSESS) 23-25 Sept. ,2015

[33] Aynur Abdurazic , Jeff offut "Coupling Based Class Integration and Test Order" AST '2006 *International Workshop on Automation of Software Test 2006'* USA

[34] Michela Pedroni and Bertrand Meyer "Object-Oriented Modeling of Object-Oriented Concepts A Case Study in Structuring an Educational Domain" http://link.springer.com/chapter/10.1007/978-3-642-11376-5_15#page-1

[35] Sujata Khatri, Dr. R. S. Chhillar and Mrs Arti sangwan "Analysis of Factors Affecting Testing in Object Oriented Systems" *International Journal of Computer Science & Engineering* Page(s)1191-1196 Vol. 3 Issue. 03.

[36] Muhammad Rabee Saheen and Lydie du Bousquet, "Relation Between the Depth of Inheritance Tree and Number Of Methods to Test" *International*

Conference on Software Testing , Verification, and Validation' 2008.

[37] Gagandeep Makkar, Jitender kumar Chh abra and Rama Krishna Challa, "Object Oriented Inheritance Metric Reusability Perspective" *International Conference on Computing, Electronics and Electrical technologies*[ICCEET], *2012*

[38] Nasib S. Gill and Sunil Sikka, "Inheritance Hierarchy based Reuse & Reusability Metrics in OOSD" *International Journal on Computer Science and Engineering (IJCSE)* Vol. 3 No. 6 June 2011.

[39] R. Harrison, S. Counsell, and R. Nithi "Experimental Assessment of the Effect Of Inheritance on the Maintainability of Object Oriented System" *Journal of System and Software* Volume 52 Issue 2 -3 , June 1 2000 pages 173-179

[40] John Daly, Andrew Brooks, James Miller, Marc Roper and Murray Wood "An Empirical Study Evaluating Depth of Inheritance on the Maintainability of Object Oriented Software" *Empirical Software Engineering* Volume 1, Issue 2, 1996, pp 109-132

[41] Arti Chhikara, R. S. Chhillar and Sujata Khatri " Evaluating the Impact of Different types of Inheritance on the Object Oriented Software Metrics" *International Journal of Enterprise Computing and Business Systems* ISSN (Online) : 223-8849 Volume 1 Issue 2 July 2011

[42] Mary Jean harrold and John D. McGreger, "Incremental Testing of object Oriented Class Structure" In *Proceedings of the 14th international Conference on software Engineering* 1992, 68-80.

[43] Gregory Seront ,Migual Lopez , Valerie Paulus and Naji Habra, "On the relationship between the cyclomatic complexity and the degree of object orientation*" aszt.inf.elte.hu*

[44] John D. McGregor Brian A. Malloy and Robecca L. Siegmund, "A Comprehensive Program Representation of Object Oriented Software" *Annals of Software Engineering* 1996, volume 2. Issue 1, pp 51 – 91

[45] Loren Larson and Mary Jean Harrold, "Slicing Object Oriented Program" International Conference on software Engineering – ICSE, pp 495- 505

[46] Anand Krishnaswamy, "Program Slicing : An application of Object oriented

Program Dependency Graphs" *11/1995 Source: CiteSeer psu.edu*

[47] David P. Tegarden, Steven D. Sheetz and David E. Monarchi, "A Software Complexity Model of Object-Oriented Systems" *Decisions Support System* 13(3-4), 1995, 241-262

[48] Santosh kumar swain ,subhend ukumar pani and Durga Prasad Mahapatra, "Model based Object-Oriented Software Testing" *Journal of Theoretical and Applied Information Technology.* 2010, 30- 36.

[49] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima and Cris Chen, "A Test Strategy for Object-Oriented Programs" *COMPSAC* 1995 : 239 – 244

[50] Bernhard rumpe, 'Model based testing of Object-Oriented Systems' *FMCO, Volume 2852 of Lecture Notes in Computer Science,* 2002, page 380-402, Springer

[51] Pranshu Gupta and David A. Gustafson "Analysis of the Class Dependency Model for Object Oriented Faults" *International Journal of Advances in Engineering & Technology,* May 2012, ISSN: 2231 – 1963

[52] Mahfuzul Huda, Arya Y.D.S., Khan M.H. "Evaluating Effectiveness Factor of Object Oriented Design: A Testability Perspective" In: *International Journal of Software Engineering & Application (IJSEA),* Vol. 6, No. 1, January (2015)

[53] Malviya Anil Kumar and Singh Vibhooti "Some Observation on Maintainability Estimation Model for Object Oriented software in requirement , Design , Coding and Testing Phases" *International Journal of Advanced Research in Computer Science and Software Engineering* Volume 5, Issue 3, ISSN: 2277 128X ,March (2015).

[54] Dinesh Kumar Saini, "Security Concern of Object Oriented Software Architectures" *International Journal of Computer Applications* (0975 – 8887) Volume 40 – No. 11, February 2012.

[55] Bremananth R and Thushara R "Fault Predictions in Object Oriented Software" *International Journal on Computer Science and Engineering* Vol. 1(2), 2009, 81-88

[56] Parvinder Singh Sandhu and Gurdev Singh, "Dynamic Metrics for Polymorphism in Object Oriented Systems" *World Academy of science,*

Engineering and Technology 15, 2008.

[57] Victor R. Basili, Lionel Briand and Walcelio L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators' *Technical Report , Univ. of Maryland, Dep. Of Computer Science, College Park, MD*, 20742 USA April 1995.

[58] Seyyed Mohsen Jamali, "Object Oriented Metrics"*SVM Header Parse 0.2*, 2006

[59] Amarnath Singh et. al, "Rephrasing Essentials of Object Oriented Programming based on Testing Pre – requisites" *(IJCSIT) International Journal of Computer Science and Information Technologies*, Vol. 2(5) , 2011, 2055 – 2059

[60] Chhikara Arti Chhikara and R.S Chhillar "Analyzing Complexity of Java Program Using Object Oriented Software Metrics" *IJCSI International Journal of Computer Science* Issues, Vol. 9, Issue 1, No.3 ISSN :1694-0814, January 2012.

[61] Bruntink Magiel and Deursen Arie Van.: Predicting Class Testability using Object Oriented Metrics" *4th IEEE International Workshop on Source Code Analysis and Manipulation*, 2004, pp .136-145, Chicago, IL, US.

[62] Ravinder kr. Gupta , Hari ji and Gajender Singh Chandel " A Fault based Object Oriented Testing using UML" *International Journal of Scientific & Engineering Research* volume 3, Issues 5, May 2012.

[63] Xiaolan Wang, Yanshuai Zhang, Hong He, "Method of the Object Oriented Program Exact Testing" *Proceedings of the Second Symposium International Computer Science and Computational Technology*" (ISCCST'09) Huangshan, P. R. China, 26 – 28 , Dec. 2009, PP 039 – 044

[64] Juliana Georgieva, Veska Gancheva, "Functional testing of Object - Oriented Software" *International Conference of Computer System and Technologies – CompSysTech* ' 2003.

[65] Nirmal Kumar Gupta and Mukesh Kumar Rohil, "Using Genetic Algorithm for Unit Testing of Object-Oriented Software" *First International Conference on Emerging Trends in Engineering and Technology* July 16 – July 18, 2008, pp 308- 313

[66] E.S.F. Najumudheen , Rajib Mall and Debasis Samanta, “A Dependence Representation for Coverage Testing of Object Oriented Programs” *Journal of Object Technology ETH Zurich, Chair of Software Engineering,* @ JOT 2010

[67] Ranjita Kumar Swain, Prafulla Kumar Behera and Durga Prasad Mohapatra “Generation and Optimization of Test Cases for Object Orinetd Software using State Chart Diagram” http://arxiv.org/ftp/arxiv/papers/1206/1206.0373.pdf

[68] David C. Kung , jerry Gao and pei Hsia, “Class, Firewall , Test Order, and Regression Testing of Object Oriented Programs’. *JOOP 8(2)* ,1995, 51 – 65.

[69] Tarun Dhar Diwan and Ganesh Suryavanshi, “Automatic Test Case Generation in Object Oriented Programming” *International Journal of Electronics and Computer Science Engineering*. ISSN 2277 – 1956, 2012

[70] Chhabi Rani Panigrahi and Rajib Mall, “A Hybrid Regression Test Selection Technique for Object Oriented Software” *International Journal of Software Engineering and its Applications* Vol. 6, No. 4, October 2012.

[71] Gregg Rothermel and Mary Jean Harrold, “Selecting Regression Tests for Object -Oriented Software” Proc. Of the *Int’l.conf. on Software Maintenance, Victoria, CA,* September 1994, pages 14 – 25.

[72] Gregg Rothermel, Mary Jean Harrold and JeinayDedhia, “Regression Test Selection for C++ Software” *Journal of Software Testing, Verification, and Reliability,* Vol. 10, no. 2, June 2000.

[73] Alessandro Orso, Nanjuan Shi and Mary Jean Harrold, “Scaling Regression Testing to Large software System” *Proceedings of the12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering* pages 2004, 241 – 251,

[74] Yanping Chen, Robert L. Probert and D Paul Sims, “Specification Based Regression Test selection with Risk analysis” *CASCON’ 02 Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research,* 2002

[75] Sheng Huang , Zhongjie li , and Jun Zhu , Yanghua Xiao and Wei Wang, “A Noval Approach to Regression Test Selection for J2ee Applications” *27th IEEE International Conference on Software Maintenance (ICSM)* 2011

[76] SubhrakantaPanda andDurga Prasad Mohapatra, “Application of Hierarchical

Slicing to Regression Test Selection of Java Programs” *Infosys Labs Briefings* VOL 11 NO 2 2013.

[77] Samaila Musa, Abu Bakar M. D. Sultan, Azim Abd Ghani and Dr. Salmi Bahrom. “Regressiong testing framework based on extended system Dependence graph for object oriented programs” *Proc. of the intl. Conf. On Advances in Computer Science & Electronics Engineering – CSEE* 2014.

[78] David Binkley “The Application of Program Slicing to Regression Testing” by, *Computer Science Department, Loyola College in Maryland*, 1998

[79] S. k. mondal and H. Tahbildar “ Regression Test Case Minimization for Object Oriented Programming using New Optimal Page Replacement Algorithm” *International Journal of Software Engineering and its Applications* vol.8 no.6 (2014).

[80] Sapna P G and Arunkumar Balakrishnan “An Approach for Generating Minimal Test Cases for Regression Testing” *Procedia Computer Science* 47 ,2015,188 – 196

[81] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. “Test case prioritization: An empirical study”. *In Proceedings of the International Conference on Software Maintenance, Oxford,* September, 1999

[82] Mohammad Rava , Wan M. N. Wan- Kadir “A Review on Prioritization Techniques in Regression Testing” *International Journal of Software Engineering and Its Applications* Vol. 10, No. 1 ,2016, pp. 221-232

[83] Sun-Woo Kim, John A Clark and John A McDermid, “Assessing Test Set Adequacy for Object- Oriented Programs using Class Mutation” *28 JAIIO Symposium on Software Technology* (SOST), 1999, pages 72-83

[84] Ranjita kumara swain , Prafulla kumar Behera and Durgaparsad mohapatra, “Minimal Test case generation for object software with state chart” *International Journal of Software Engineering & Application* (IJSEA), Vol. 3 No.4, 2013.

[85] Ranjita Kumari Swain, Prafulla Kumar Behera and Durga Prasad Mohapatra “Generation and Optimization of Test Cases for Object Oriented Software using State Chart Diagram” *arXiv preprint arXiv: 1206.0373*,2012, arxiv.org

[86] Chhabi Rani Panigrahi and Rajib Mall, “A Heuristic – based Regression Test Case Prioritization Approach for Object Oriented Programs” *Innovations Syst. Softw Engg* DOI 10 ,1007 / 11334 – 013 – 0221, 2014 pages 155-163.

[87] J. A. Parejo, Ana B. Sanchez, S. Sagura, A. R. Corets, Robert E. Lopez-Herrejon, and Alexander Egyed “Multi- Objective Test Case Prioritization Technique For Highly Configurable Systems: A Case Study” *The Journal Of System Software 122,*2016, 287 -310

[88] J. Ding , X. Yi Zhang “ Comparison Analysis Of Two Test Case Prioritization Approaches With The Core Idea Of Adaptive”*29th Chinese Control And Decision Conference (CCDC),*2017

[89] R. Haung, J. Chen, D. Towey, Alvin T. S. Chan, Yansheng Lu “Aggregate-Strength Interaction Test Suite Prioritization” *The Journal Of System And Software* 99,2015, 36-51

[90] Hao Dan, Zhang Lingming, Zhang Lu, Rothermel Gregg and Mei Hong “ A Unified Test Case Prioritization Approach” *ACM Transactions on Software Engineering and Methodology* , Vol. 9, No.4 Article 39, 2010

[91] Vincenzo Martena , Alessandro Orso and Mauro Pezze, “Interclass Testing of Object Oriented Software” *ICECCS IEEE Computer Society,* 2002, page 135 – 144.

[92] Muhammad Shahid & Suhaimi Ibrahim “A New Code based Test Case Prioritization Technique” *International Journal of software Engineering and its Application* Vol. 8, No 6 , 2014, PP. 31 – 38.

[93] R. Beena & S. Sarala “Code coverage based test case selection and prioritization” *International Journal of Software Engineering and Applications* (IJSEA), Vol.4, No. 6 2013

[94] Alessandro Marchetto **,** Md. Mahfuzul Islam and Waseem Asghar **“**A Multi-Objective Technique to Prioritize Test Cases” *IEEE Transactions on Software Engineering* , Volume 20 Issue 10 2016

[95] Prakash. N and Rangaswamy T. R “ Weighted Method For Coverage Based Test Case Prioritization” *Journal of Theoretical and Applied Information Technology 20th October 2013. Vol. 56 No.2 © 2005 - 2013 JATIT & LLS.*

[96] K. Uma Maheshwar and S. Vasundra " Automated Functional Test Case Prioritization For Increased Rate Of Fault Detection" *International Journal for Innovative Research in Science & Technology| Volume 1 | Issue 7 | December 2014 ISSN (online): 2349-6010*

[97] Preeti and S. Bishnoi " Test Case Prioritization Technique for Object Oriented Software based On Source Code Analysis" *International Journal of all Research Education and Scientific Method* volume 4 issue 6 ,2016.

[98] Ajay Kumar Jena, Santosh Kumar Swain, Durga Prasad Mohapatra "Test Case Generation And Prioritization Based On Uml Behavioral Models" *Journal of Theoretical and Applied Information Technology* Vol.78, 2015.

[99] Zengkai Ma and Jianjun Zhao "Test Case Prioritization based on Analysis of Program Structure" *15th Asia-Pacific Software Engineering Conference,* 2018

[100] Chhabi Rani Panigrahi and Rajib Mall "Test Case Prioritization for Object Oriented Programs" *Set labs Briefings* Vol 9, No 4, 2011.

[101] Patwa Sanjeev and Malviya Anil Kumar "Impact of Coding Phase on Object Oriented Software Testing" *Covenanat Journal of Informatics and Communication Technology(CJICT)* Vol.2, No. 1, June, 2014.

[102] Mitrabinda Ray and Durga Prasad Mohapatra "Code-based prioritization: a pre-testing effort to minimize post-release failures" *Innovations Syst Softw Engg*8 : 279 -292 DOI 10.1007/s 11334 -012 – 0186- 3, 2012.

[103] R. Krishnamoorthy and S. A. Saahaya Arul Mary, "Factor Oriented Requirement Coverage Based System Test Case Prioritization of New and Regression Test Cases", *International Journal of Information Software Technology,* Vol. 51, No. 4, pp. 799–808, 2009.

[104] R. Kavitha , N. Sureshkumar "Test Case Prioritization for Regression Testing based on Severity of Fault**"** (IJCSE) *International Journal on Computer Science and Engineering* Vol. 02, No. 05, 2010, 1462-1466

[105] E. Ashraf, A. Rauf, and K. Mahmood "Value based Regression Test Case Prioritization" *Proceedings of the World Congress on Engineering and Computer Science* 2012 Vol. I WCECS 2012, October 24-26, , San Francisco, USA, 2012

[106] Arup Abhinna Acharya, Sonali Khandai And Durga Prasad Mohapatra "A Novel approach for test case prioritization using Business criticality test value" *International Journal of Computer Applications (0975 – 8887)* Volume 46– No.15, May 2012.

[107] T. Muthusamy and Seetharaman. K " A New Effective Test Case Prioritization for Regression Testing based on Prioritization Algorithm" *International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868 Foundation of Computer Science FCS, New York, USA* Volume 6– No. 7, January 2014 – www.ijais.org

[108] S. Nayak, C. Kumar and S. Tripathi " Enhancing Efficiency Of The Test Case Prioritization Technique By Improving The Rate Of Fault Detection" *Arab Journal of Science and engineering* © King Fahd University of Petroleum & Minerals 2017.

[109] Hyuncheol Park, Hoyeon Ryu and Jongmoon Baik "Historical Value-Based Approach for Cost-cognizant Test Case Prioritization to Improve the Effectiveness of Regression Testing" *Second International Conference on Secure System Integration and Reliability Improvement* 2008

[110] R. Kavitha and N. Sureshkumar "Requirement based Test Case Prioritization with Equal Weightage for factors" *International Conference on Mathematical Computer Engineering - ICMCE* – 2013

[111] Monika Tayagi and Sona Malhotra "An Approach for Test Case Prioritization Based on Three Factors" *I.J. Information Technology and Computer Science,* 2015, 04, 79-86

[112] Thillaikarasi Muthusamy and Dr. Seetharaman.K "A Test Case Prioritization Method with Weight Factors in Regression Testing Based on Measurement Metrics" *International Journal of Advanced Research in Computer Science and Software Engineering* Volume 3, Issue 12, December 2013.

[113] Sahar Tahvili, Wasif Afzal, Mehrdad Saadatmand, Markus Bohlin, Daniel Sundmark and Stig Larsson "Towards Earlier Fault Detection by Value-Driven Prioritization of Test Cases Using Fuzzy TOPSIS" *Advances in Intelligent Systems and Computing* 448, Springer International Publishing Switzerland

2016.

[114] Everton L. G. Alves , Patricia D.L. Machado, Tiago Massoni and Miryung Kim “Prioritizing Test Cases for Early Detection of Refactoring Faults *Software Testing , Verification and Reliability* Volume 26 Issue 5, 2016.

[115] Md. Junaid Arafeen and Hyunsook Do “Test Case Prioritization Using Requirements-Based Clustering” *IEEE Sixth International Conference on Software Testing, Verification and Validation* 2013.

[116] H. Sarikanth, M. Cashman and M. B. Cohen “ Test Case Prioritization Of Build Acceptance Tests For An Enterprise Cloud Application: Industrial Case Study” *The journal of system and software 119,* 2016 *122- 135*

[117] Debasish Kundu et.al, “ System Testing for Object Oriented System With Test Case Prioritization” *Journal Software Testing, Verification & Reliability* volume 19 issue 4, December, 2009.

[118] Sangeeta Sabharwal , RituSibal and Chayanika Sharma, “Applying Genetic Algorithm for prioritization of Test Case Scenarios Derived from UML diagrams” *IJCSI International Journal of Computer Science* Issues, Vol. 8, Issue 3, No. 2, May 2011 ISSN 1694 – 0814.

[119] Chhabi Rani Panigrahi , Rajib Mall “A Heuristic-based Regression Test Case Prioritization Approach for Object-Oriented Programs” *Innovation System Software Engineering* , 2014 10:155 -163 DOI 10.1007/s

[120] Samaila Musa, Abu-Bakar Md Sultan, Abdul-Azim Bin Abd-Ghani and Salmi Baharom “Software Regression Test Case Prioritization for Object-Oriented Programs using Genetic Algorithm with Reduced-Fitness Severity” *Indian Journal of Science and Technology,* Vol 8(30), DOI: 10.17485/ijst/2015/v8i30/86661, 2015, ISSN (Print) : 0974-6846

[121] Fevzi BELL , M¨ubariz ˙ , Bekir Taner “Model-Based Test Case Prioritization using Cluster Analysis: A Soft-Computing Approach” *Turkish Journal of Electrical Engineering & Computer Sciences Turk J Elec Eng & Comp Science* ,2015

[122] A. Bakar Md sultan, .A. Ghani, S. Baharom and S. Musa. “An Evolutionary Regression Test Case Prioritization Based on Dependence Graph and Genetic

Algorithm for Object Oriented Programs" *2nd International Conference on Emerging Trends in Engineering and Technology*, May 30-31 , 2014 London(UK).

[123] P. Saraswat, A. Singhal et.al. " A Hybrid Approach For Test Case Prioritization And Optimization Using Meta – Heuristics Techniques" *978-1-4673-6984-8/16/ © 2016 IEEE 2016*

[124] S. Mahajan, S. D. Joshi and V. Khanna "Component Based Software System Test Case Prioritization With Genetic Algorithm Decoding Technique Using Java Platform" *2015 International Conference On Computing Communication Control And Automation*, 2015

[125] S. Ghai, S. Kaur " A Hill Climbing Approach For Test Case Prioritization" *International journal of Software engineering and its Applications Vol. 11, No. 3 , 2017, pp. 13-20* http://dx.doi.org/10.14257/ijseia.2017.11.3.0.

[126] S. Kumar mohapatra and srivinas Prasad " Test Case Reduction using Ant Colony Optimization for Object Oriented Programs" *International Journal of Electrical and Computer Engineering* vol. 5 no.6 ,2015

[127] S. Raju, G. V. Uma 'Factors Oriented Test Case Prioritization Technique in Regression Testing using Genetic Algorithm" *European Journal of Scientific Research ISSN 1450-216X Vol.74 No.3 (2012), pp. 389-402 © EuroJournals Publishing, Inc*. 2012 http://www.europeanjournalofscientificresearch.com

[128] Rijwan Khan, Mohd amjad "Automatic Test Case Generation Of Test Cases For Data Flow Test Path Using K Mean Clustering And Genetic Algorithm" *International Journal of Applied Engineering Research* ISSN 0973-4562 Volume 11, Number 1 ,2016, pp 473-478

[129] Ahlam Ansari, Anam Khan, Alisha Khan, Konain Mukadam "Optimized Regression Test Using Test Case Prioritization" Proceeding of Computer Science Volume 79, 2016, pages 152-160

[130] Erum Ashraf , Tamim Ahmed Khan , Khurrum Mahmood and Shaftab Ahmed "Value based PSO Test Case Prioritization Algorithm" *International Journal of Advanced Computer Science and Applications*, Vol. 8, No. 1, 2017.

[131] Gregg Rothermel , Roland H.Untch , Chengyun Chu , Mary Jean Harrold

"Prioritizing Test Cases for Regression Testing" IEEE Transactions on Software Engineering Volume: 27, Issue: 10, Oct 2001.

[132] Wasiur Rhmann**,** Taskeen Zaidi and Vipin Saxena "Test Cases Minimization and Prioritization Based on Requirement, Coverage, Risk Factor and Execution" Time *British Journal of Mathematics & Computer Science* 14(1): 1-9, 2016, Article no.BJMCS.23269

[133] M. Yoon, Eunyoung Lee, M. Song, B. Choi " A Test Case Prioritization through Correlation of Requirement and Risk" *Journal of Software Engineering and Applications, 2012, 5, 823-835 http://dx.doi.org/10.4236/jsea.2012.510095 Published Online October 2012 (http://www.SciRP.org/journal/jsea)*

[134] C. Hettiarachchi, H. Do, B. Choi et.al. "Risk Based Test Case Prioritization Using A Fuzzy Expert System" *Information And Software Engineering 69,2016 1- 15*

[135] W. Rahman, V. Saxena " Fuzzy Expert System Based Test Case Prioritization From UML State Machine Diagram Using Risk Information" *I.J. Mathematical Sciences and Computing, 2017, 1, 17-27 Published Online January 2017 in MECS (http://www.mecs-press.net) DOI: 10.5815/ijmsc.*2017.

[136] Hema Srikanth, Charitha Hettiarachchi, Hyunsook Do "Requirements Based Test Prioritization Using Risk Factors: An Industrial Study" *Information and Software Technology* , 2015, doi: 10.1016/j.infsof.2015.09.002

[137] HojunJaygarl, Kai-Shin Lu, Carl K. Chang, "GenRed: A Tool for Generating and Reducing Object-Oriented Test Cases' COMPSAC' 10': *IEEE International Computer Science and Applications Conference,*(Seoul, Korea), 2009

[138] Ming - Chi Lee Tamkang 'An Object oriented Testing Framework Specified in Z Notation' *Journal of Science and Engineering* , vol. 2 No. 1 PP. 11-22 1999

[139] Tao Xie, KunalTaneja , Shreyas Kale and Darko Marinov, "Towards a framework for differential Unit testing of object- oriented Programs" *Proceeding AST '07 Proceeding of the Second International Workshop on Automation of Software Test* page , 2007.

[140] Taewoong Jeon, Hyon Woo Seung and Sungyoung Lee, "Embedding Built in Tests in Hot Spots of an object oriented Framework" *ACM SIGPLAN Notices*

Volume 37 Issue 8, August 2002 pages 25-34

[141] Jehad Al Dallal, “Testing Object Oriented Framework Hook Methods” *Kuwait Journal of Science & Engineering*, 2008.

[142] Taweesup Apiwattanapong , Alessandro Orso, Maty Jean Harrold “J Diff: A Differencing Technique and Tool for Object Oriented Programs” *Autom Software Engg(2007)* 14:3 – 36 DOI 10. 1007/s 10515 – 006 – 0002 – 0 springer science Business Media, LLC 2006.

[143] Amie L. Souter, Tiffany M. Wong, Stacey A Shindo, Lori L. Pollock , ‘ “TATOO : Testing and analysis Tool for Object Oriented Software” *TACAS* 2001 : 389 – 403.

[144] Yu Xia Sun , Huo Yan Chen , “A new approach and CASE tool for object-oriented dynamic tests at cluster-level with data types of pointer and reference” *7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software*, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings pp 389-4032001 10.1007/3-540-45319-9_27 Springer Berlin Heidelberg.

[145] Jitenedra S. Kushwaha and Mahendra S. Yadav “Testing for object oriented software” *Indian Journal of Computer Science and Engineering* (IJCSE).

[146] Anna Derezinska , Anna Szustek, “Object Oriented Testing Capabilities and Performance Evaluation of C# Mutation System” *CEE – SET* ,2009 , 229 – 242

[147] Christian Engel , Christoph Gladisch, Vladimir Klebanov and Philipp Rummer, “Integrating Verification and Testing of Object Oriented Software” *TAP*, 2008,182-191

[148] Bor – Yuan Tsai , Simon Stobart , Norman Parrington and Ian Mitchell, “A State based Testing Approach Providing Data Flow Coverage in Object Oriented Class Testing*” The 12th International Software Quality Week Conference 1999(QW 99)* , May 1999, San Jose USA

[149] Recardo Terra and Macro Tulio “A Dependency Constraints Language to Manage Object Oriented Software Architectures” *Software Practice and Experience*, 2009; 39: 1073–1094p. Willy inderscience DOI: 10.1002/spe.931.

[150] Hyunsook Do, Gregg Rothermel and Alex Kinneer, "Empirical Studies of Test Case Prioritization in a J Unit testing Environment" *Digital Object Identifier : 10.1109/ISSRE.2004.18* PublicationYear : 2004, Pages (s): 113-124

[151] Nicolas Frechette, Linda Badri and Mourad Badri, "Regression Test Reduction for Object Oriented Software : A Control Call Graph based Technique and Associated tool" *Hindawi Publishing Corporation ISRN Software Engineering* Volume 2013 , Article ID 420394, 10 pages

[152] http://cppprojectcode.blogspot.in/2010/09/income-tax-calculation.html.

[153] http://cppprojectcode.blogspot.com/2010/09/football-players-position-information.html

[154] http://www.cppforschool.com/project/super-market-billing.html

[155] https://github.com/

[156] H. Kumar, vedpal, N. chauhan " A Unit – Test Case prioritization Technique based on source code analysis" *International journal of Advanced Research in computer science and software engineering* Volume 5, Issue 4, 2015

[157] Vedpal and N. Chauhan "A Multi factor Coverage based Test case Prioritization Technique for object oriented software" *International Journal of System and Software Engineering* Volume 3 Issue 1 2015.

[158] http://cppprojectcode.blogspot.in/2010/09/dispensary-management-system.html

[159] https://projectabstracts.com/1690/payroll-management-system-in-java.html

[160] https://www.ibm.com/products/spss-modeler

[161] http://www.public.iastate.edu/~kkoehler/stat557/tree14p.pdf

[162] Reeta Sahoo *C++ Projects*. Delhi : Khanna Book Publishing Co. (P) LTD. 2000

[163] G. Rothermel, M,j. Harrold, "A Safe Efficient Regression Test Selection Technique," ACM Transactions on software engineering Methodology 1997; 6(2): 173–210p.

[164] HesterDecoz "Capgemini World quality report 2014" www.worldqualityreport.com

[165] http://www.ece.ubc.ca/~matei/EECE417/BASS/ch09lev1sec6.html

[166] C. Zhao and Roger T. Alexander, "Testing AspectJ Programs using Fault-Based Testing", *Workshop on Testing Aspect Oriented Programs (WTAOP'07),*

Vancouver, British Columbia, Canada, ACM, 2007

[167] https://www.d.umn.edu/~gshute/softeng/object-oriented.html

[168] https://en.wikipedia.org/wiki/Programming_language

[169] S. Malhotra and S. Chaudhary " Programming in Java" *Oxford University Press*

[170] ftp://public.dhe.ibm.com/software/analytics/spss/support/Stats/Docs/Statistics/Algorithms/13.0/TREE-QUEST.pdf

[171] https://cran.r-project.org/web/packages/C50/vignettes/C5.0.html

[172] https://www.statsoft.com/Textbook/Classification-and-Regression-Trees

[173] https://en.wikipedia.org/wiki/Fragile_base_class

[174] S.R. Chidamber , C.F. Kemerer "A metrics suite for object oriented design" *IEEE Transactions on Software Engineering* Volume: 20, Issue: 6, Jun 1994

APPENDIX A

Source code for Performing Simple Banking Functions

```
class account
{
char cust_name[20];
int acc_no;
char acc_type[20];
public
void get_accinfo()
{
cout<<”\n enter customer name:-“;
cin>>acc_no;
cout<<”enter account type:-“;
cin>>acc_type;
}
void display_accinfo()
{
cout<<”\ncustomer name:-“<<cust_name;
cout<<”\n account number:-“<<acc_no;
cout<<”/n account type:-“<<acc_type;
}
};
class cus_account: public account
{
static float balance;
public:
void disp_cusbal()
{
cout<<”/n balance:-“<<balance;
}
void deposit_cusbal()
```

```
{
float deposit;
cout<<"\n enter amount to deposit";
cin>>deposit;
balance=balance+deposit;
}
void withdraw_cusbal()
{
float penalty,withdraw;
cout<<"\n balance:-"<<balance;
cout<<"\n enter amount to withdraw"
cin>>withdraw;
balance=balance-withdraw;
if(balance<500)
{
penalty=(500-balance)/10;
balance=balance-penalty;
cout<<"\n balance after deducity penalty:"<<balance;
}
elseif(withdraw>balance)
{
cout<<"/n you have to take permission for bank overdraft facility";
balance=balance+withdraw;
}
else
cout<<"/n after withdrawl your balance reveals:"<<balance;
}
};
class sav_account: public account
{
static float sav_bal;
```

```
public:
void disp_savbal()
{
cout<<”\n balance:-“<<savbal;
}
void deposit_savbal()
{
float deposit,interest;
cout<<”\n enter amount to deposit:-“;
cin>>deposit;
savbal=savbal+deposit;
interest=(savbal*2)/100;
}
void withdraw_savbal()
{
float withdraw;
cout<<”\n balance:-“<<savbal;
cout<<”\n enter amount to withdraw:-“;
cin>>withdraw;
savbal=savbal-withdraw;
if(withdraw>savbal)
{
cout<<” you have to take permission for bank overdraft facility\n”;
savbal=savbal+withdraw;
}
cout<<”\n after withdraw your balance “,<<savbal;
}
};
float cus_acct||balance;
float sav_acct||savbal;
void main()
```

```
{
clrscr();
cus_acct c1;
sav_acct s1;
cout<<"\enter s for saving customer and c for current account customer\n";
char type;
cin>>type;
int choice;
if(type=='s' ||type=='s')
{
s1.get_acc_info();
while(1)
{
clrscr()
cout<<"/n choose your choice";
cout<<"1)deposit\n";
cout<<"2)withdraw\n";
cout<<"3)display balance\n";
cout<<"4)display with full detail\n";
cout<<"5)exit\n";
cout<<"6)choose your choice:-";
cin>>choice;
switch(choice)
{
case 1: s1.deposit_savbal();
getch();
break;
case 2: s1.withdraw_savbal();
getch();
break;
case 3:s1.disp_savbal();
```

```
getch();
break;
case 4: s1.display_accinfo();
s1.disp_savbal();
getch();
break;
case 5: goto end;
default!cout<<”by year”;
}
}
}
else
{
{
c1.get_accinfo();
while(1)
{
cout<<”\n choose your choice\n”;
cout<<”1)deposit”;
cout<<”2)withdraw”;
cout<<”3)display balance”;
cout<<”4)display with full details”;
cout<<”5)exit”;
cout<<”6)choose your choice:’;
cin>>choice;
switch(choice)
{
case 1: c1.deposit_cusbal();
getch();
break;
case 2: c1.withdraw_cusbal();
```

```
getch();
break;
case 3: c1.disp_cusbal();
getch();
break;
case 4: c1.display_accinfo();
c1.disp_cusbal();
getch();
break;
case 5: goto end;
default : “cout<<try again”;
}
}
}
end
}
}
```

APPENDIX B

Source Code to Calculate the Income Tax

```
package org.j2eedev.calc;

import java.lang.*;
import java.io.*;
class Employee
{

        String name;
        String des;
        int pay;
        int gp;
        char posting;
        public int hra;
        public int cpf;
        int tax_inc;



        public void getdata() throws IOException
        {
                System.out.println("Pleae ented the details");
                BufferedReader br = new BufferedReader (new
InputStreamReader(System.in));
                name=br.readLine();
                System.out.println("Pleae ented the designation");
                des = br.readLine();
                System.out.println("Pleae ented payscale");
                pay = Integer.parseInt(br.readLine());
                System.out.println("Pleae ented gradepay");
```

```
            gp = Integer.parseInt(br.readLine());
            System.out.println("Pleae ented posting( M/NM");
            posting = (char)br.read();

        }
            public void display()
        {
            //System.out.println ("Name of Student: "+name);
            System.out.println ("Name of the employee: "+name);
            System.out.println ("Designation of the employee"+des);
            System.out.println ("Pay scale of the Employee: "+pay);
            System.out.println ("grade pay of the employee "+gp);
            System.out.println ("Posting of the employee "+posting);
        }
            class temp_employee
            {
                int salary1 = 10000;
                 int get_salary()
                {
                    return salary1;
                }
                 void print()
                 {
                     System.out.println ("Salary of the employee "+salary1);
                 }
            }
    }
    class Result extends Employee
    {

        /*public void gross()
        {
                int total= pay+gp;
                float percent=total*100/200;
                System.out.println ("Percentage: "+total+"%");
        }*/
        public void display()
        {
                super.display();
        }
}
class salary extends Employee
{
        int g_salary,da,total;
public void gross() throws IOException
{
        super.getdata();

        System.out.println(" gross salary is" +posting );
        if(posting == 'M')
        {
                total = pay+gp;
                hra = (20*total)/100;
                cpf = (10*total)/100;
                da = total;
                g_salary = total+hra+da;
                //System.out.println(" gross salary is" +g_salary );
        }
                else
                                        {
                                total = pay+gp;
                                hra = (20*total)/100;
                                cpf = (10*total)/100;
                                da = total;
                                g_salary = total+hra+da;



                        }
            }
        public void display()
        {
            System.out.println(" gross salary is" + g_salary);
                        System.out.println(" HRA is" + hra);
                        System.out.println(" CPF" + cpf);
        }
            }
class deduction extends Employee
{
            void total_deduction() throws IOException
            {
                        //super.gross();
                        //int total = pay + gp;
                        //int hra = 20*total/100;
                        int t_hra = 12 * hra;
                        //int cpf = 10*total/100;
                        int t_cpf = 12*cpf;
                        int t_ded = t_hra +t_cpf;
                        System.out.println(" total deduction of employee" +t_ded);


            }
}
class saving extends Employee
```

```
{


	void tex_saving() throws IOException
	{
		//super.getdata();
		//super.display();
		BufferedReader kr = new BufferedReader (new
InputStreamReader(System.in));
		System.out.println("enter the ammount acc to 80 C");
		int am1 = Integer.parseInt(kr.readLine());
		System.out.println("enter the ammount acc to 80 d");
		int am2 = Integer.parseInt(kr.readLine());
		System.out.println("enter the ammount acc to mediclaim");
		int am3 = Integer.parseInt(kr.readLine());
		int total_saving = am1+am2+am3;
		System.out.println("total savings" +total_saving);

	}
}

class tax_cal extends Employee
{
	void tex() throws IOException
	{
		super.getdata();
		BufferedReader tr = new BufferedReader (new
InputStreamReader(System.in));
		System.out.println("plz enter the annula income");
		int inc = Integer.parseInt(tr.readLine());
		System.out.println("plz enter the deduction");
```

```
                int ded = Integer.parseInt(tr.readLine());
                System.out.println("plz enter the annual Saving");
                int sav = Integer.parseInt(tr.readLine());
                System.out.println("plz enter the sex of employee M?F");
            char ab = (char) tr.read();

            if (ab == 'M')
            {
                int t_sav = 200000+ ded+sav;
                tax_inc = inc - t_sav;

                }
            else
            {
                int t_sav = 200000+ ded+sav;
                tax_inc = inc - t_sav;
            }

        }
        public void display()
         {
                 System.out.println(" income under the tex is"+tax_inc);
         }
}
class tax_paid extends tax_cal
{
        int tex;
        void paid() throws IOException
        {
                tex();
                try
            {
            if(tax_inc >= 500000)
            {
                   tex = (20*tax_inc)/100;
                  //System.out.println("total tex to be paid" +tex);
            }
            else
            {
                   tex = (10*tax_inc)/100;
                  //System.out.println("total tex to be paid" +tex);
            }
            }
            catch( Exception exp)
            {
            System.out.println(exp.toString());
            }
      }

      public void display()
      {
            System.out.println("total tex to be paid" +tex);
      }
      {

      }

}



/*class Result1 extends Student
{
        Result1(String n, int r, int m1, int m2)
        {
                super(n,r,m1,m2);
        }
        public void sum()
        {
                int total=(mark1+mark2);
                System.out.println ("Percentage: "+total+"%");
        }
        void display()
        {
                super.display();
        }
}*/

public class Multiple
{
        public static void main(String args[]) throws IOException
        {



                                System.out.println("*************************");
                System.out.println("press1");
                System.out.println("press2");
                System.out.println("press3");
                System.out.println("press4");
                System.out.println("press5");
                System.out.println("press6");
                System.out.println("*************************");
                BufferedReader br = new BufferedReader (new
InputStreamReader(System.in));
```

```
int choice = Integer.parseInt(br.readLine());
Result R = new Result();
salary s = new salary();
deduction d = new deduction();
saving s1 = new saving();
tax_cal t = new tax_cal();
tax_paid t1 = new tax_paid();
Employee e = new Employee();
Employee.temp_employee f = e.new temp_employee();
switch(choice)
{
        case 1:

                R.getdata();
                R.display();
        case 2:

                s.gross();
                s.display();
        case 3:
                s.gross();
                d.cpf = s.cpf;
                d.hra = s.hra;
                d.total_deduction();
        case 4:
                s1.getdata();
                s1.tex_saving();
        case 5:

                t.tex();
        case 6:
```

```
                    t1.paid();
                    t1.display();

                case 7:
                f.print();




            }


        }

    }
```

APPENDIX C

Survey to Determine the Weight of Proposed Factors

Resources of Survey

In this survey 123 software professional participated working in various software industries. The people being surveyed are the project managers, lead technology, team lead, QA, Tester. The average experience of the software professional is approximate 8 years.

About Survey

The conducted survey is based on the factors related to the object oriented programming language. The respondents were asked to assigned a positive weight to each factor. The assign weight shows the capability to introduce the errors in the software. The sum of assigned weights to factors should be 1. This survey was conducted to identify where usually the errors are introduces and propagate from one level to another level. The following Table C.1 Shows the Considered Factors.

Table C.1 Considered Factors

S.No.	**Factors**
1	Class/Interface
2	Type Casting
3	Exception handling
4	Method Overloading
5	Native Method
6	Nested Class
7	Conditional Statements
8	Number of methods

Sample of Survey Form

Name _________________ Company Name______________________

On the basis of the criticality of the factor a weight is assigned to the factors given in table. The assigned weight shows the capability of introducing the error in the software.

(1) The value of weight should be between the 0 and 1.

(2) The sum of the weights assigned to all factors should be 1

S. No	**Factor**	**Weight**	**Agree**	**Neutral**	**Disagree**	**If disagree then assign weight according to you**
1	Class/Interface	.05				
2	Type Casting	.15				
3	Exception Handling	.3				
4	Method Overriding	.2				
5	Native Method	.1				
6	Nested Class	.05				
7	Conditional Statements	.05				
8	Number of Method	.1				

All respondents participated indicated that the considered factors can affect the testing of any software. By using the considered factors the testing of the software is very effective and helps to reduce the cost of testing of software. A majority of the respondents were agreed with the assigned value as shown below in the Table C.2. Other respondents have partially different view abut assigned weight to the factors

Table C.2 : Weight Assigned to Proposed Factors

S. No.	Factors	Weight
1	Class/Interface	0.5
2	Type Casting	0.15
3	Exception handling	.3
4	Method Overloading	.2
5	Native Method	.1
6	Nested Class	0.5
7	Conditional Statements	0.5
8	Number of methods	.1

The Result of analysis of survey of the all considered factors is shown in Figures From C.1 to C.8

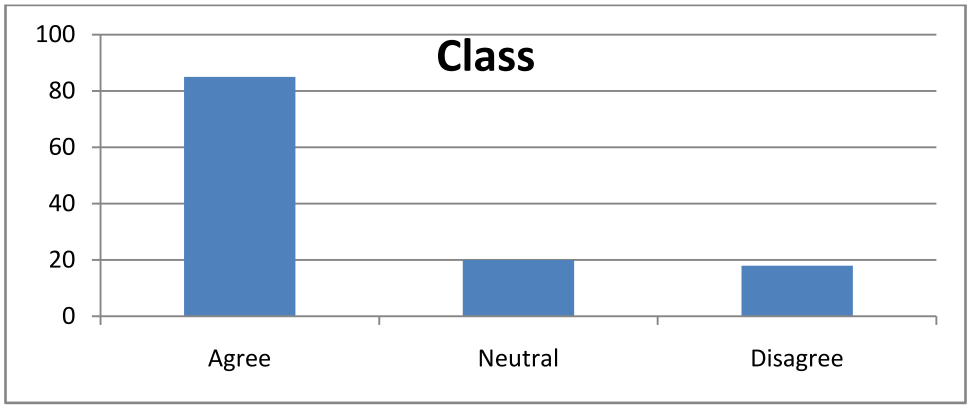


Figure C.1: Responses for Factor Class

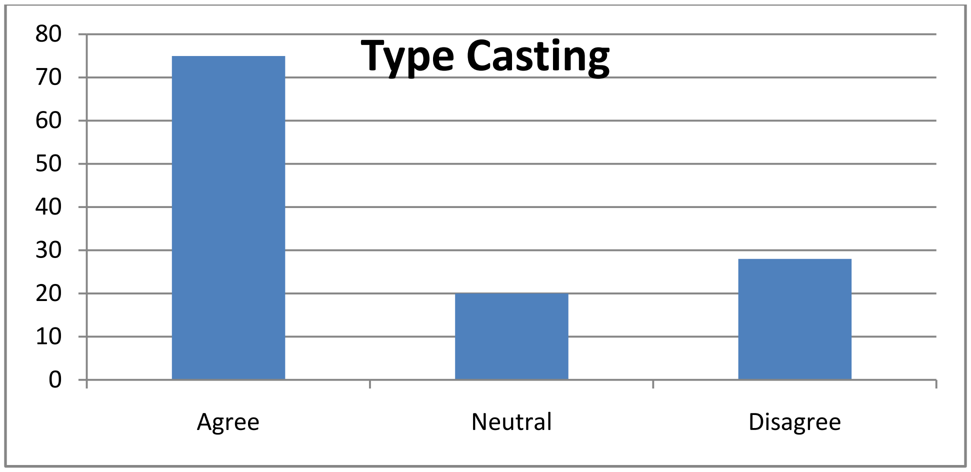


Figure C.2: Responses for Factor Type Casting

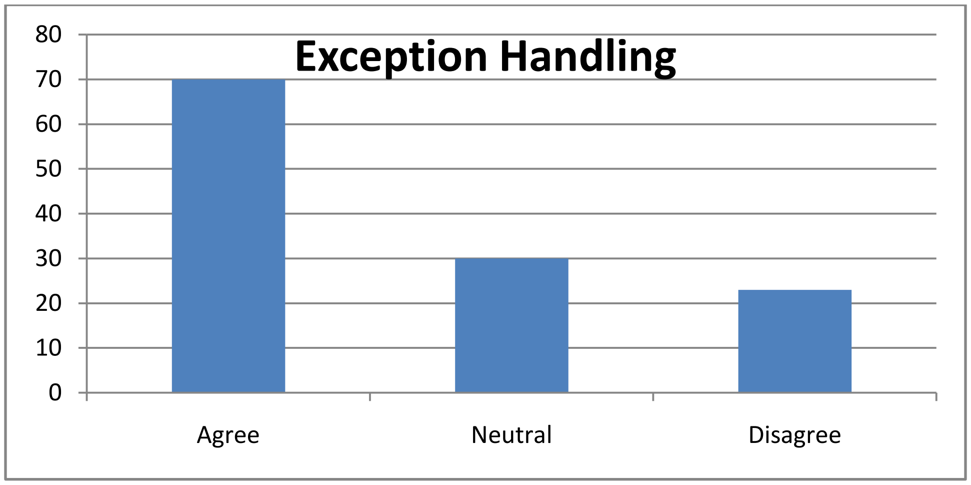


Figure C.3: Responses for Factor Exception Handling

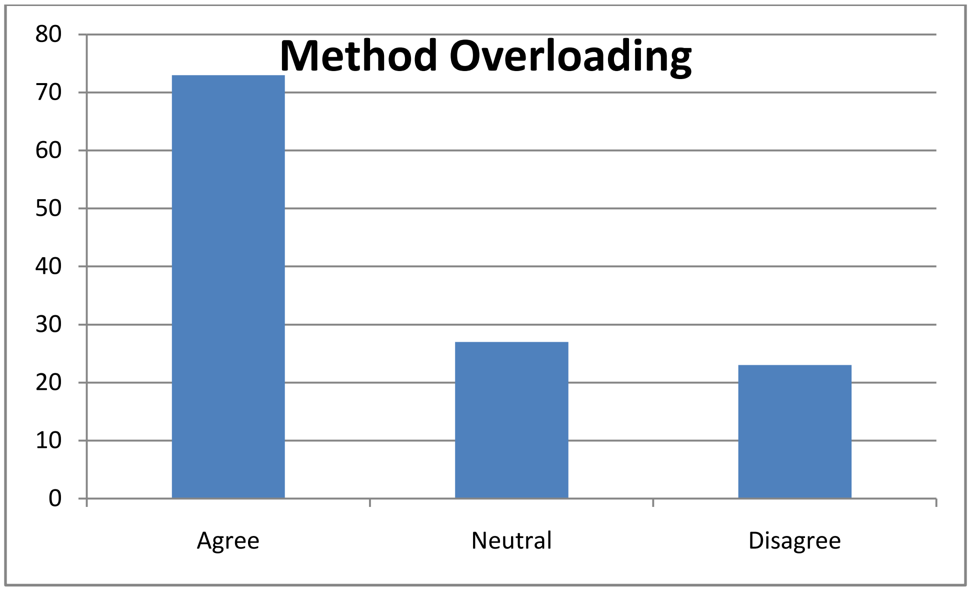


Figure C.4: Responses for Factor Method Overloading

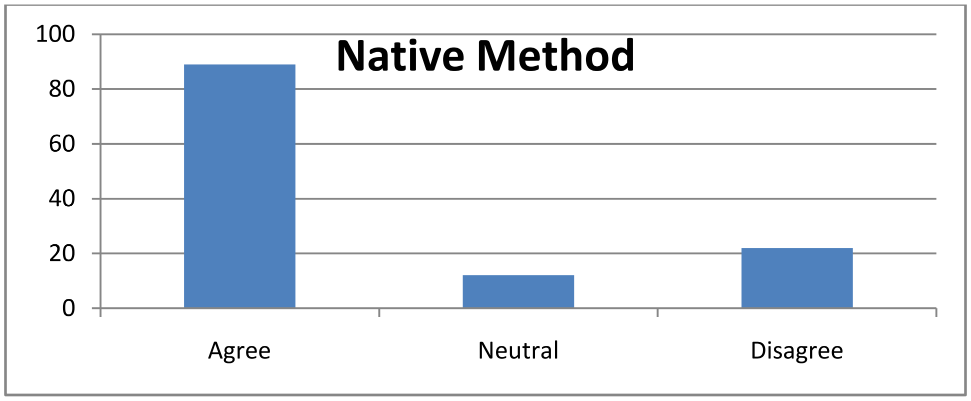


Figure C.5: Responses for Factor Native Method

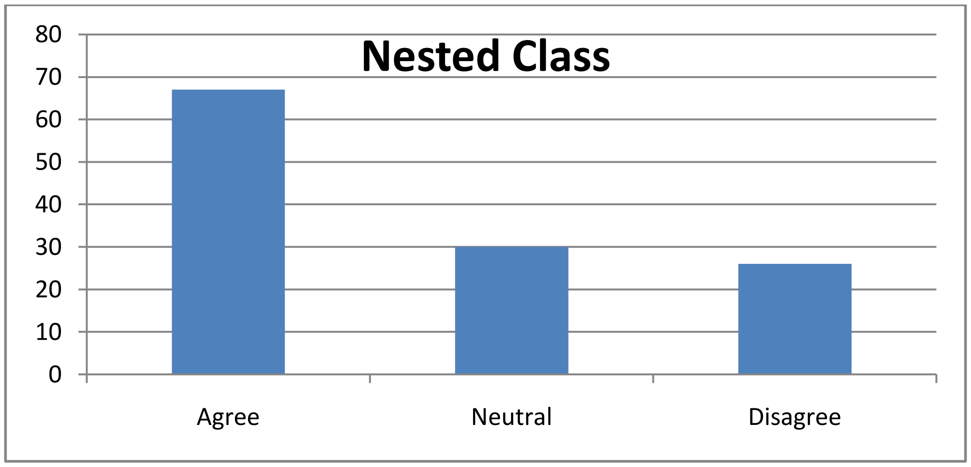


Figure C.6: Responses for Factor Native Method

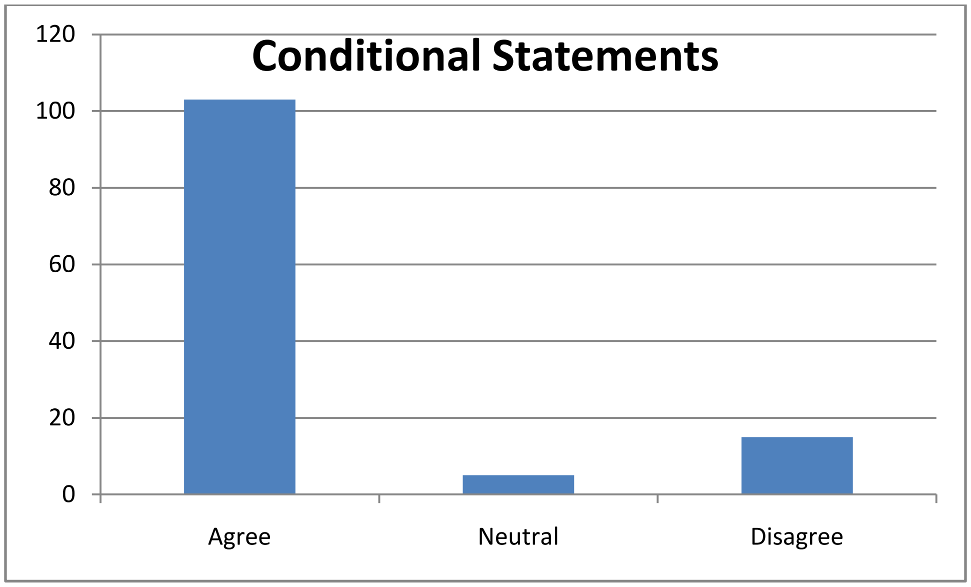


Figure C.7: Responses for Factor Conditional Statements

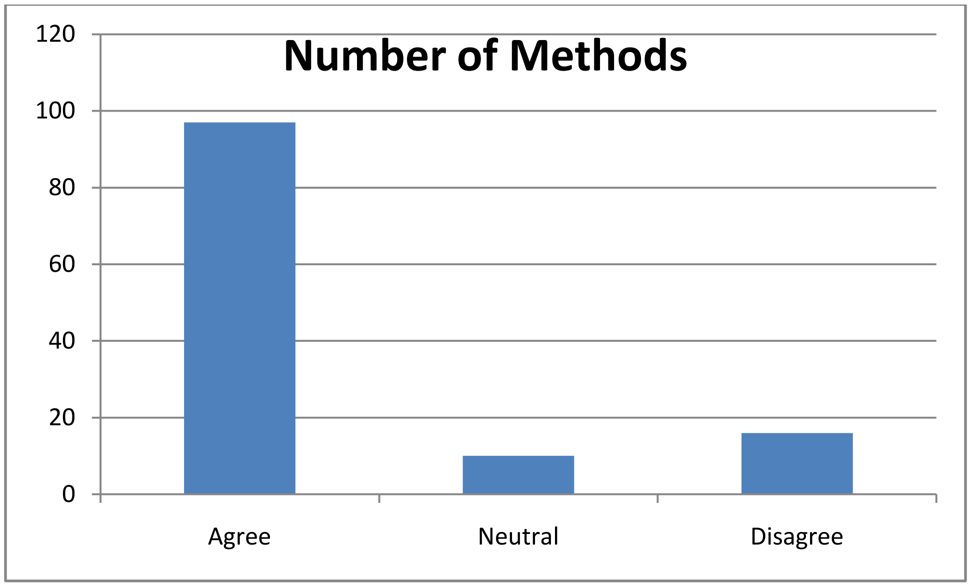


Figure C.8: Responses for Factor Conditional Statements

APPENDIX D

Survey to check the viability of some factors and assigned weight

Test case prioritization is a process to order test cases with the intention of finding maximum faults as earlier as possible. Prioritization of the test cases is performed on the basis of some factors. In this survey some factors are considered to prioritize the test cases. Every considered factor has been assigned a positive weight within range of the 0 to 1 which shows the probability to introduce the error in the object oriented software if the developer did not use it in right way. So you are requested to assign a weight that suits to you on the basis of your experience. The Table D.1 shows the questionnaire of the survey.

The Result of Survey analysis is given in figure 6. In the given figure weight ranges are representing by the slabs as given below

Slab1 = 0=<Wt<.0.3

Slab2 = 0.3=<Wt<.0.5

Slab3 = 0.5=<Wt<0..8

Slab4 = 0.8=<Wt=1

Table D.1 Questionnaire of Performed Survey

S.No	**Factor Name**	**Slab 1**	**Slab 2**	**Slab 3**	**Slab 4**
1	Degree of Method(DM)				
2	No. of Input variable (VU)				
3	Decision statement (DS)				
4	Type Casting(TC)				
5	Numerical computations (NC)				
6	Number of loop(LS)				
7	Number of variable reused (VR)				
8	Copying of objects (CO)				
9	Object/Data reads from				

	database/File(RW)				
10	Exception handling (EH)				
11	Virtual function (VF)				
12	Dynamic memory allocation and deallocation (MA)				
13	Reference counting (RC)				
14	Proxy Objects (PO)				
15	Type binded inherited Function (TIF)				
16	Copy constructor having pointer type variable (CPV)				
17	Non virtual destructor (NVD)				
18	Return object by reference (RO)				

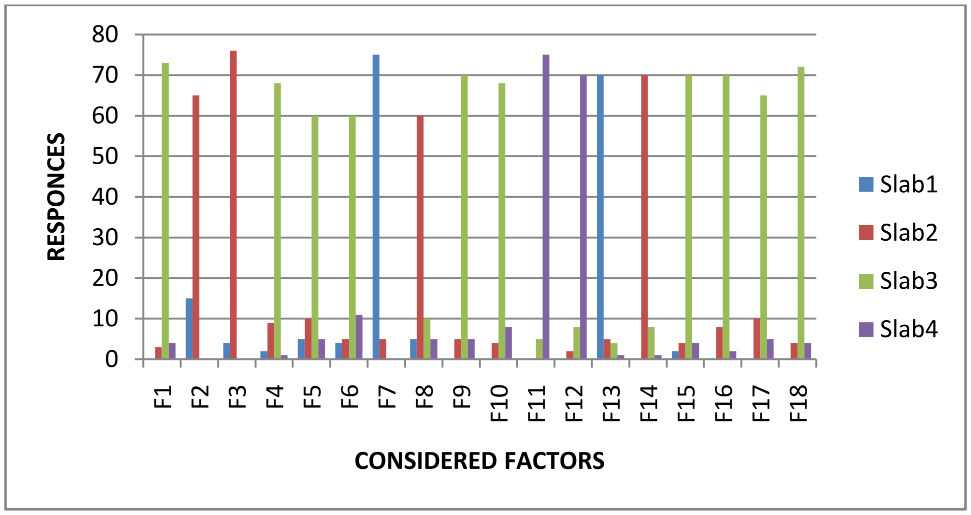


Figure D.2. Analysis of Feedback from Participations

The weight is determined by the calculating the mean average of the weight assigned by the participants.

APPENDIX E

The snapshots of the working of SPSS Modeler is Shown in Figure E.1 to E.6

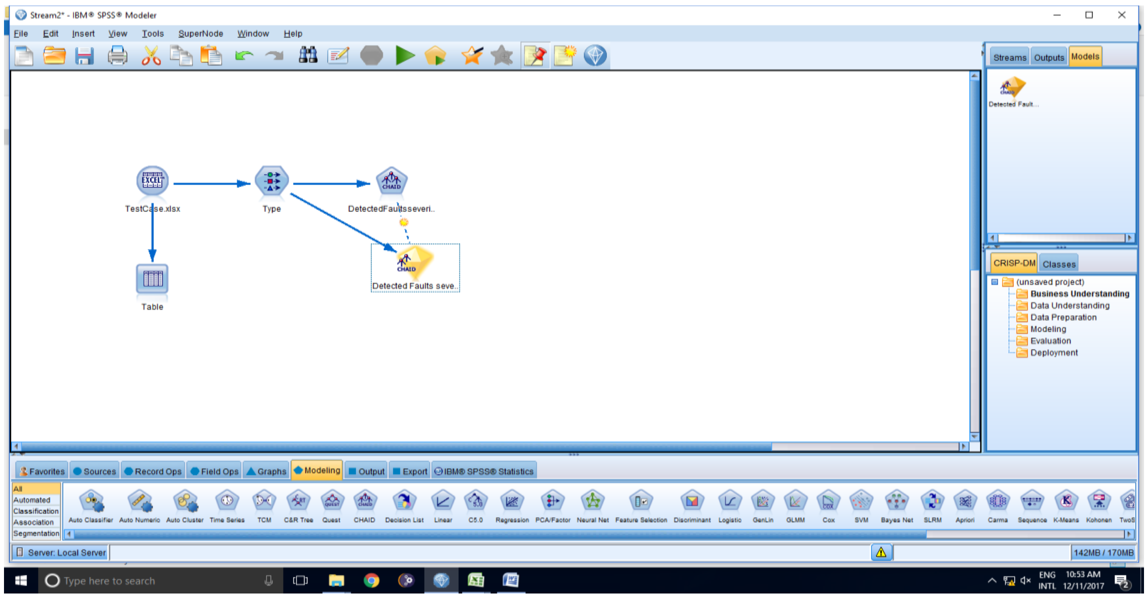
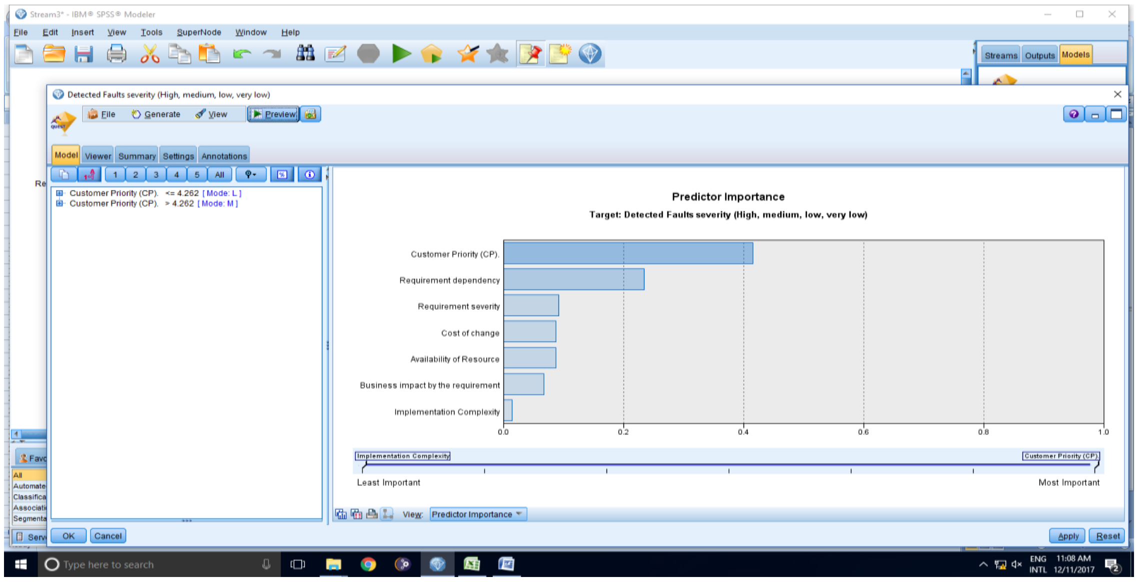

Stream2* - IBM® SPSS® Modeler
File Edit Insert View Tools SuperNode Window Help
Streams Outputs Models
Detected Fault...
TestCase.xlsx
Type
DetectedFaultsseveri..
Detected Faults seve..
Table
CRISP-DM Classes
(unsaved project)
Business Understanding
Data Understanding
Data Preparation
Modeling
Evaluation
Deployment
Favorites Sources Record Ops Field Ops Graphs Modeling Output Export IBM® SPSS® Statistics
All
Automated
Classification
Association
Segmentation
Auto Classifier Auto Numeric Auto Cluster Time Series TCM C&R Tree Quest CHAID Decision List Linear C5.0 Regression PCA/Factor Neural Net Feature Selection Discriminant Logistic GenLin GLMM Cox SVM Bayes Net SLRM Apriori Carma Sequence K-Means Kohonen TwoS
Server: Local Server
142MB / 170MB
Type here to search
ENG INTL 10:53 AM 12/11/2017

Figure E.1: Process to Obtain the Contribution Weight to Prioritized the Requirements


Stream3* - IBM® SPSS® Modeler
File Edit Insert View Tools SuperNode Window Help
Streams Outputs Models
Detected Faults severity (High, medium, low, very low)
File Generate View Preview
Model Viewer Summary Settings Annotations
1 2 3 4 5 All
Predictor Importance
Target: Detected Faults severity (High, medium, low, very low)
Customer Priority (CP).
Requirement dependency
Requirement severity
Cost of change
Availability of Resource
Business impact by the requirement
Implementation Complexity
0.0 0.2 0.4 0.6 0.8 1.0
Implementation Complexity
Customer Priority (CP)
Least Important
Most Important
View: Predictor Importance
OK Cancel
Apply Reset
Type here to search
ENG INTL 11:08 AM 12/11/2017

```
Customer Priority (CP).   <= 4.262 [ Mode: L ]
Customer Priority (CP).   > 4.262 [ Mode: M ]
```

Figure E.2: Determined Contribution Weight of Requirement Factors

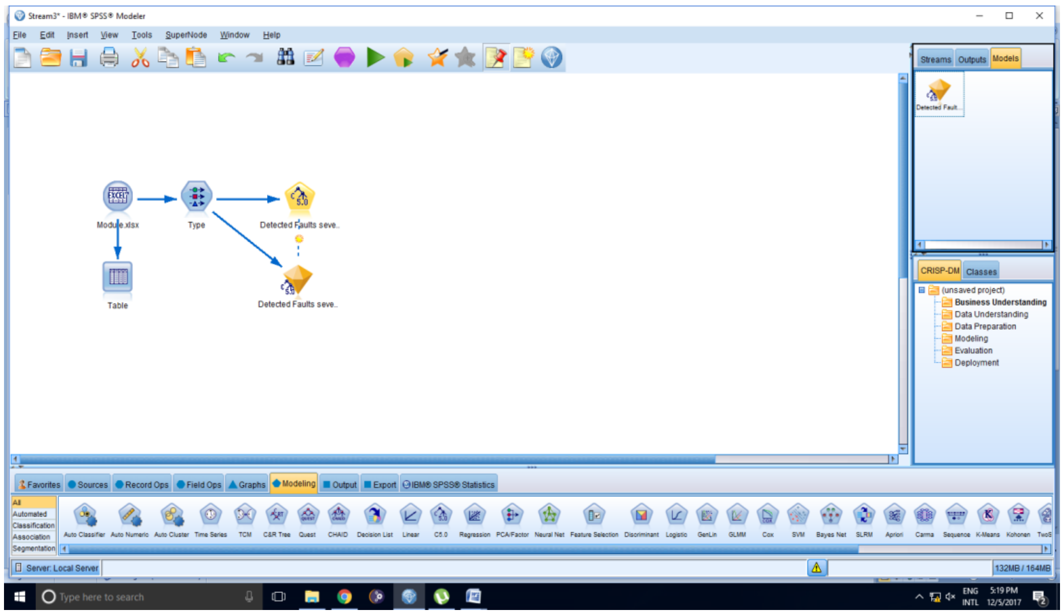
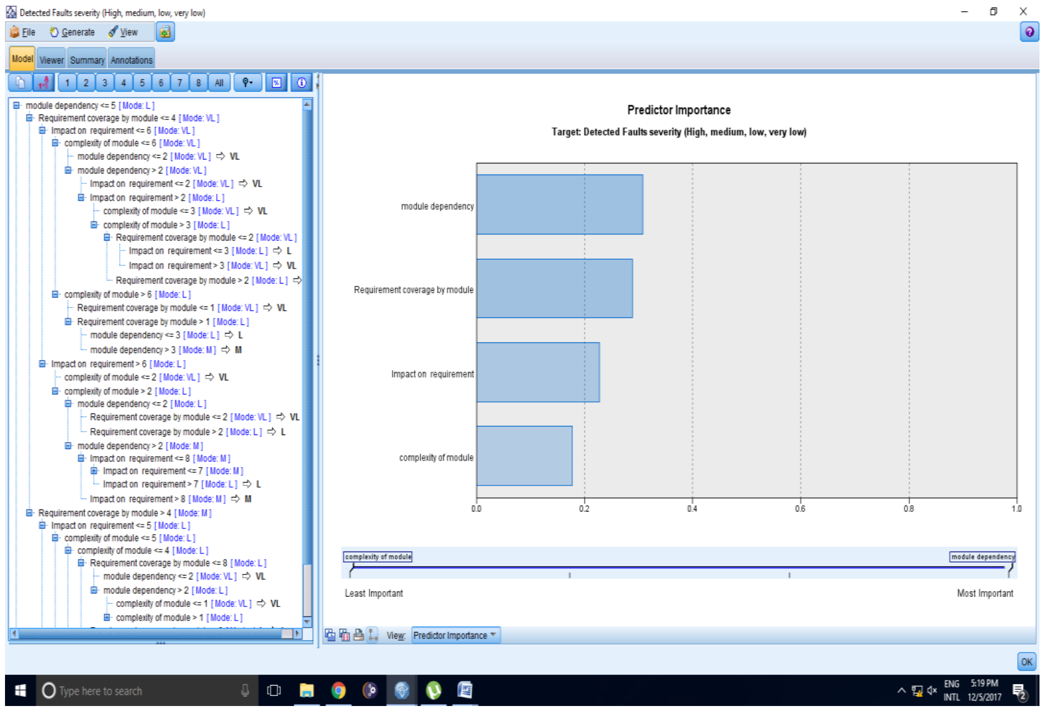


Figure E.3: Process to Obtain the Contribution Weight to Prioritized the Modules


Detected Faults severity (High, medium, low, very low)
File
Generate
View
Model
Viewer
Summary
Annotations
Predictor Importance
Target: Detected Faults severity (High, medium, low, very low)
module dependency
Requirement coverage by module
Impact on requirement
complexity of module
0.0
0.2
0.4
0.6
0.8
1.0
Least Important
Most Important
View: Predictor Importance
OK

```
⊟ module dependency <= 5 [ Mode: L ]
  ⊟ Requirement coverage by module <= 4 [ Mode: VL ]
    ⊟ Impact on requirement <= 6 [ Mode: VL ]
      ⊟ complexity of module <= 6 [ Mode: VL ]
          module dependency <= 2 [ Mode: VL ] ⇨ VL
        ⊟ module dependency > 2 [ Mode: VL ]
            Impact on requirement <= 2 [ Mode: VL ] ⇨ VL
          ⊟ Impact on requirement > 2 [ Mode: L ]
              complexity of module <= 3 [ Mode: VL ] ⇨ VL
            ⊟ complexity of module > 3 [ Mode: L ]
              ⊟ Requirement coverage by module <= 2 [ Mode: VL ]
                  Impact on requirement <= 3 [ Mode: L ] ⇨ L
                  Impact on requirement > 3 [ Mode: VL ] ⇨ VL
                Requirement coverage by module > 2 [ Mode: L ] ⇨
      ⊟ complexity of module > 6 [ Mode: L ]
          Requirement coverage by module <= 1 [ Mode: VL ] ⇨ VL
        ⊟ Requirement coverage by module > 1 [ Mode: L ]
            module dependency <= 3 [ Mode: L ] ⇨ L
            module dependency > 3 [ Mode: M ] ⇨ M
    ⊟ Impact on requirement > 6 [ Mode: L ]
        complexity of module <= 2 [ Mode: VL ] ⇨ VL
      ⊟ complexity of module > 2 [ Mode: L ]
        ⊟ module dependency <= 2 [ Mode: L ]
            Requirement coverage by module <= 2 [ Mode: VL ] ⇨ VL
            Requirement coverage by module > 2 [ Mode: L ] ⇨ L
        ⊟ module dependency > 2 [ Mode: M ]
          ⊟ Impact on requirement <= 8 [ Mode: M ]
            ⊞ Impact on requirement <= 7 [ Mode: M ]
              Impact on requirement > 7 [ Mode: L ] ⇨ L
            Impact on requirement > 8 [ Mode: M ] ⇨ M
  ⊟ Requirement coverage by module > 4 [ Mode: M ]
    ⊟ Impact on requirement <= 5 [ Mode: L ]
      ⊟ complexity of module <= 5 [ Mode: L ]
        ⊟ complexity of module <= 4 [ Mode: L ]
          ⊟ Requirement coverage by module <= 8 [ Mode: L ]
              module dependency <= 2 [ Mode: VL ] ⇨ VL
            ⊟ module dependency > 2 [ Mode: L ]
                complexity of module <= 1 [ Mode: VL ] ⇨ VL
              ⊞ complexity of module > 1 [ Mode: L ]
```

Figure E.4: Determined Contribution Weight of Requirement Factors

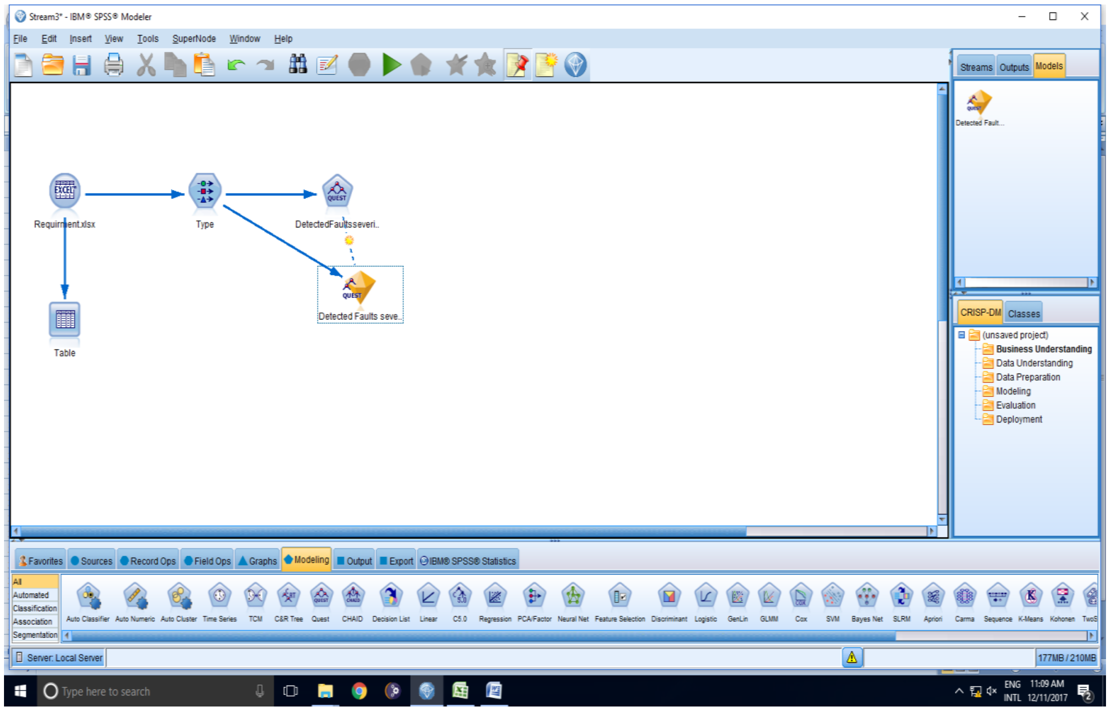
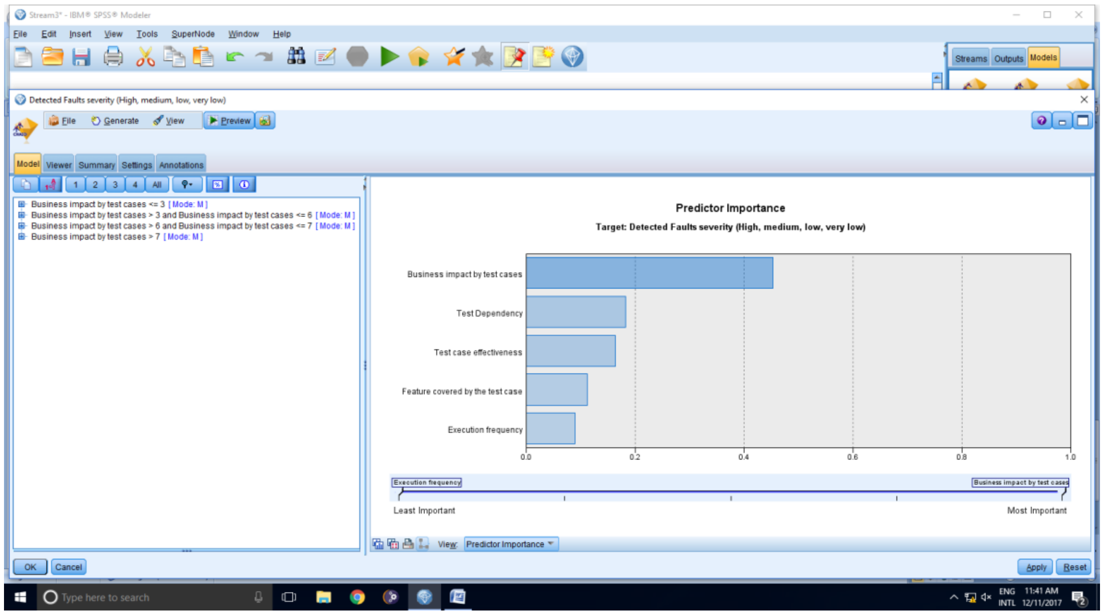


Figure E.5: Process to Obtain the Contribution Weight to Prioritized the Requirements


Stream3* - IBM® SPSS® Modeler
Detected Faults severity (High, medium, low, very low)
Model
Viewer
Summary
Settings
Annotations
Predictor Importance
Target: Detected Faults severity (High, medium, low, very low)
Business impact by test cases
Test Dependency
Test case effectiveness
Feature covered by the test case
Execution frequency
0.0
0.2
0.4
0.6
0.8
1.0
Least Important
Most Important
View: Predictor Importance
OK
Cancel
Apply
Reset

Business impact by test cases <= 3 [Mode: M]
Business impact by test cases > 3 and Business impact by test cases <= 6 [Mode: M]
Business impact by test cases > 6 and Business impact by test cases <= 7 [Mode: M]
Business impact by test cases > 7 [Mode: M]

Figure E.6: Determined Contribution Weight of Requirement Factors

APPENDIX F

Survey to Determine the Prioritized Regression Test Cases: This survey is performed to obtain the weight to various factors with the objective to prioritize the regression test cases on the basis of the surveyed factors. The factors are related to the past history of the testing of the software. Every participant was asked to assign the positive weight in the range of 0 to 1. The weight is assigned on the basis of the capability of the factors to determine the maximum faults as earlier as possible. The Table F.1 shows the considered factors.

Table F.1: Factors Related to the Past history of Testing

S.no.	**Factor Name**
1	Severity of Bug
2	Capability of Detecting the Bug
3	Coverage of Code
4	Impact on business
5	Execution Time

The 85 Participations have participated in the survey. The Figure F.1 shows the analysis of the received responses from the participants.

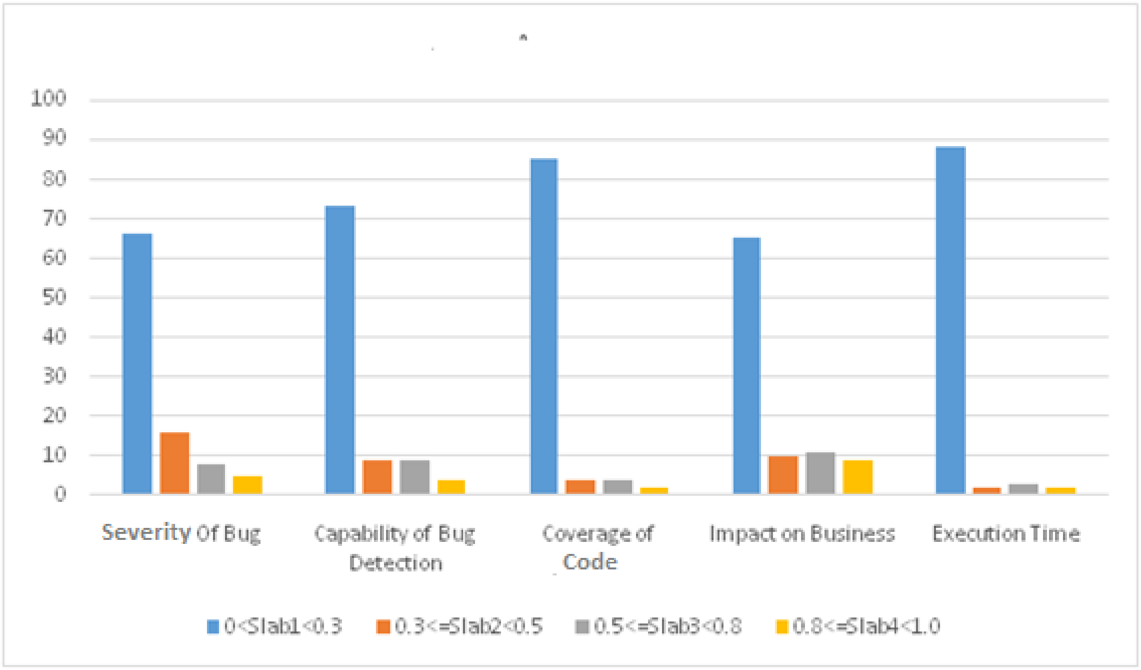


Figure F.1: Analysis of Received Responses from Participations

The Figure F.2 shows the determined weight of the factors by analyzing the received responses from the participants.

Table F.2: Determined Weight of the Factors

S.no.	**Factor Name**	**Factor Weight**
1	Severity of Bug	.25
2	Capability of Detecting the Bug	.2
3	Coverage of Code	.15
4	Impact on Business	.3
5	Execution Time	.1

BRIEF BIO DATA OF RESEARCH SCHOLOR

Vedpal is pursuing his Ph.D. in Computer Engineering from J. C. Bose University of Science & Technology, YMCA, Faridabad, M. Tech (CE) From YMCA University of Science & Technology, Faridabad in year 2012, MCA from MD University Rohtak in year 2008. He has nine years of experience in teaching. Presently he is working as an Assistant Professor in department of Computer Applications in J. C. Bose University of Science & Technology, YMCA, Faridabad, Haryana, India. His Research Areas include Software engineering, Software Testing and Object Technology.

LIST OF PUBLICATIONS OUT OF THESIS

(I) List of Published Papers in International Journals

S. No.	Title of Paper	Name of Journal where Published	No.	Volume & Issues	Year	Pages
1	Object Oriented Testing: Review and Analysis". ISSN 2014	International Journal of Engineering Research & Informatics	2348 - 6481	Vol. 1 Issue 6	2014	
2	A Multi - Factor coverage based Test Case Prioritization Technique for Object Oriented Software". ISSN Number: 2015	International Journal of System and Software Engineering	2321-6017	Volume 3 Issue 1	2015	18-23
3	A Fault – Severity based Regression Test Case Prioritization Technique for Object Oriented Software" ISSN : Nov 2016	International Journal of Computer Science Engineering (IJCSE)	2319-7323	Vol. 5 No.06	2016	312-326
4	Test Case Prioritization Technique for Object Oriented Software	International Journal of Innovative Computing,	ISSN 1349-4198	Volume 14 Number 1	2018	341-354

	Using Method Complexity	Information and Control(IJCIC)				
5	A Technique for Regression Testing of Object Oriented Software” ,	Asian Journal of computer Science and Technology	ISSN 2249 – 0701	Volume 7 number 1	2018	87-92

(II) List of Published Papers in National Journals

S. No .	Title of Paper	Name of Journal where Published	No.	Volume & Issues	Year	Pages
1	CORFOOS : Cost Reduction Framework for Object Oriented System	Journal of Computer Science Engineering and Software Testing	-	Volume 1 Issue 1	2015	1-13

(III) List of Communicated Papers in International Journals

S.No.	Title of Paper	Name of Journal	Present Status	Year
1	A Structural Analysis based Test Case Prioritization Technique for Object Oriented Software	International Arab Journal of Information Technology Engineering and Software Testing	Under Review	2016

2	A Coupling – Analysis based Test Case Prioritization Technique for Object Oriented Software	International Journal of Innovative Computing, Information and Control	Under Review	2018

(IV) List of Published Papers in International Conferences

S.No.	**Title of Paper**	**Name of Conference**	**Year**
1	A Hierarchical Test Case Prioritization technique for Object Oriented Software	International Conference Contemporary Computing and Informatics (IC3I) , Mysore, India (IEEE)	2014
2	Regression Test Case Selection for Object Oriented Systems Using OPDG and Slicing Technique	2nd International Conference on Computing for Sustainable Global Development, BVICAM, Delhi (IEEE)	2015

3	A Multi - Factored Cost And Code Coverage based Test Case Prioritization for Object Oriented Software	50th Golden Jubilee Annual Convention CSI, BVICAM, Delhi (Proceeding Published by Springer)	2015