CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Design and Implementation of Test case Prioritization in iValidator

A thesis submitted in partial fulfillment of the requirements

For the degree of Master of Science

in Software Engineering

By

Anusha Nataraj

May 2011

The thesis of Anusha Nataraj is approved:

_____	_____
George Wang, Ph.D.	Date
_____	_____
Robert Lingard, Ph.D.	Date
_____	_____
Shan Barkataki, Ph.D., Chair	Date

California State University, Northridge

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

SuT	System under Test
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformation
API	Application Programming Interface
TFR	Test step Failure Rate
CC	McCabe's Cyclomatic Complexity
SC	Static analysis defect Count
LOC	Lines Of Code
BCEL	Byte Code Engineering Library
GUI	Graphical User Interface
NCSS	Non Commenting Source Statements
L	List

LIST OF FIGURES

LIST OF TABLES

Abstract

Design and Implementation of test case prioritization in iValidator

by

Anusha Nataraj

Masters in Software Engineering

Software defects cost the US economy an estimated 59.5 billion dollars each year [20]. It has been suggested that improved testing, focused on by earlier identification and removal of defects, could save $22.2 billion [20]. Various techniques have been proposed to increase the success rate in early defect detection, especially during integration and regression testing. Typically, such techniques rely on executing the test cases within a test scenario in some strategic order. Results from experiments conducted using such techniques indicate some degree of success in early detection of software defects [2] [6] [11].

The objective of this work is to facilitate early detection of defects using a test ordering technique based on test case prioritization. The proposed technique works by assigning a priority to each test case in a suite of test cases and then executing the test cases in descending priority order. The priority values are computed using an algorithm developed as part of this work, the algorithm computes priorities as a function of defects detected in the previous test runs (test history), presence of error prone code constructs and McCabe's complexity. This technique is intended to be used during integration testing and regression testing.

The concepts developed in this project have been implemented and integrated into an open source test tool called “iValidator”. The iValidator tool is capable of automatically executing a suite of test cases in some specified order. It is also capable of reporting the test results and maintaining a test history. In the iValidator nomenclature, test cases are called test steps. A test step is a composite entity that includes one or more software unit to be tested and the associated unit test descriptions. A test step can relate to testing of a use case or a sequence within a use case. It can also represent a collection of test cases used in functional testing of a software component. A collection of test steps make up a test description. Typically, each test description is associated with a System under Test (SuT) representing the higher-level software application being tested.

In this work, the iValidator tool has been extended to provide capabilities for performing static code analysis for detecting error prone code constructs, and for computing McCabe complexity values. The results are expressed in XML. The enhanced tool then uses these computed values, together with the previously recorded test history to compute the test priority for each test step in a test suite. In a typical test scenario, the enhanced iValidator tool is used to determine the test priorities of a collection of test steps in a test description. After that the tool is programmed to execute the test steps in descending order of their test priorities. The tool generates two reports upon completion of each test run. The first report describes the test execution results for the test steps in the test suite. The second report describes the test execution history, results from static analysis for error prone code constructs, and the McCabe complexity values.

A prototype of the iValidator tool enhancements has been designed and implemented. The enhancements have been tested and validated with code production quality code.

1 Introduction

Software test activities are performed throughout the software development lifecycle. More than 50% of software costs and time is spent on testing the software [21]. Early detection of defects in the testing cycle can be beneficial in providing quick feedback to the team on product quality [22]. This is true as it gives developers ample time to fix the defects and retest. One approach is to identify maximum defects early by ordering test cases in a specific manner. This approach of ordering test cases is termed as "test case prioritization" [2]. Each test case is assigned a priority value in order to determine the position of the test case in an ordered list. Some of the parameters that are considered to determine the priority of the test cases are test case code coverage and test case execution history.

Test case prioritization orders test cases in a particular sequence to achieve predefined objectives. An objective can be to increase test code coverage, early detection of high severity defects, reduce testing costs and so on. The objective depends on the test plan and software development phase in which the testing is performed. The technique or algorithm to be used for ordering test cases is determined based on the objective. The benefits of test case prioritization are:

- Detection of defects early in the testing cycle
- Minimization of testing costs
- Increasing test efficiency
- Minimization of testing time

- Identification of high risk defects earlier
- Increasing reliability
- Increasing test code coverage early in the testing lifecycle
- Achieving better return on investment with more efficient time and resource utilization.

In this thesis, I present the design and implementation of a test case prioritization algorithm within a test tool called "iValidator". The algorithm is designed to prioritize test steps in iValidator. The order of test steps is determined to increase the rate of defect detection. This is achieved by identifying defect prone code based on static analysis, complexity analysis and history of test execution.

iValidator is an open source test automation tool mainly designed for integration testing but can be used for white box and black box testing [5] as well. The iValidator test infrastructure supports test script development, test execution and test result reporting. The test scripts and reports are expressed in XML. iValidator is developed in Java. The thesis extends the capability of iValidator to execute test steps in a sequence that would increase the rate of defect detection. The algorithm is implemented in Java.

This thesis is organized into 9 chapters. This chapter provides an introduction to the idea of the thesis with the desired objectives to be achieved. Chapter 2 provides a survey and critical analysis of existing literature in the field of Test case Prioritization definition and techniques. This chapter summarizes previous research studies performed on the different techniques used for prioritizing test cases. This chapter also provides an overview on

iValidator test tool, iValidator terms and definitions, architecture, class diagram, features and an example. Chapter 3 describes the use case model. Chapter 4 presents the functional requirements for implementation of test step prioritization in iValidator. Chapter 5 describes the rationale, flow and pseudo-code of the test step prioritization algorithm. Chapter 6 discusses the detailed design by presenting the modified iValidator architecture, class and sequence diagrams. This chapter also discusses the selection of tools to perform static analysis and complexity analysis in iValidator. Chapter 7 describes the implementation details. Chapter 8 provides the test rationale followed to test the extended iValidator tool. Chapter 9 summarizes the goals and objectives, benefits and limitations and future improvements.

1.1 Background

Test case prioritization orders and schedules test case executions based on computed priorities. It does not reduce the number of test cases to be executed [6] and hence does not have the disadvantage of not executing some test cases. Many techniques have been proposed to prioritize test cases based on the desired objective to be achieved. Different techniques consider different parameters to calculate priority. Some parameters are listed below [6]:

a) Code coverage

b) History of test execution

c) Cost effectiveness

d) Customer Requirements

Greedy and Genetic algorithms have been used to implement some of the test case prioritization techniques. Greedy algorithm [2] selects the optimal set of test cases. This set of test cases is determined by prioritizing the test cases having higher potential of detecting faults over other test cases. Genetic algorithm is a search evolution technique to determine optimal search results. It is used to search for optimal test case based on multiple objectives like defect detection, defect severity, test costs and many other factors [16]. Based on experiments performed using these techniques the results indicate an improvement in rate of fault detection and reduced costs [2] [6].

1.2 Problem

Although many techniques have been proposed for test case prioritization, there are some research areas that require attention [2] [6]. Some of the research areas are:

a) Utilizing static analysis to identify defects based on risky or poor coding patterns.
b) Determining order between test cases having same priority
c) Prioritizing test cases across multiple test suites where each test suite consists of multiple test cases

1.3 Objectives

The objectives of the thesis are as listed below:

a) Design an algorithm for prioritizing test steps such that the more defect prone test steps are assigned higher priorities over other test steps. This facilitates priority based ordering of the test steps to achieve the specific test objectives listed in section 1.1. This algorithm prioritizes test steps defined in a test description in

section 2.3.1. The defect prone code of the System under Test (SuT) are identified based on the below listed parameters:

1) Static analysis of source code:

   This parameter is used to identify implementation defects. This analysis focuses on identification of defects that arise from use of ambiguous or potentially erroneous code patterns [8], misuse of language semantics, dangling pointers, boundary conditions, memory leaks and buffer overflows.

2) Complexity analysis:

   This parameter is used to determine McCabe's Cyclomatic complexity.

3) Test execution history:

   This parameter is used to determine the number of times the test step has failed in prior executions.

The priority of the test step is calculated based on the above listed parameters.

b) Implement the designed test step prioritization algorithm in iValidator.

The test step prioritization algorithm is implemented as an in-built functionality within the iValidator test tool. The following capabilities will be added to iValidator to achieve this objective:

1) Ability to perform static analysis on the SuT.
2) Ability to determine complexity analysis on the SuT.
3) Ability to maintain test step execution history
4) Ability to execute test steps in the sequence determined by the test step prioritization algorithm

5) Ability to display the metrics collected for each test step such as static analysis, complexity analysis, and test step execution history.

2 Literature review

2.1 Test case prioritization

Test case prioritization orders test cases to achieve an objective, which, if the test cases were executed in any other order would not have been achieved. The definition and different techniques of test case prioritization are described below.

2.1.1 Test case prioritization definition

Test case prioritization problem as defined by Rothermel [2] is described below:

" Given: T is a set of test cases, PT the set of all possible ordering (prioritizations) of T and f a function from PT to the real numbers

Problem: Find T' ε PT such that (for all T") (T" ε PT) (T" ≠ T') [f (T') ≥ f (T")] "

T', T" represent the different ordering of test cases in test suite T. The test case prioritization problem described in the definition is to determine a permutation of T which maximizes the test objective.

f is a function that determines the success of the permutations. It assigns a quantitative value to the permutation determined on the basis of test objective to be achieved. The permutation of T with higher value awarded by f is the preferred order of test cases. The logic or algorithm to be implemented as function depends on a specific objective to be achieved. Examples of such objective can be increasing rate of defect detection or increasing code coverage or increasing the rate of detecting high severity defects. It is essential to specify the objective quantitatively. A vague test objective, like increasing the speed of testing cannot be translated into a logical test ordering procedure. The test

objective helps decide the parameters and the algorithm of test case prioritization to be implemented.

The different types of test case prioritization are [2]:

a) General test case prioritization:

The test objective is to test the program over different versions. The test case prioritization logic is to determine an ordering of test cases which could be used over subsequent versions of the program. The ordering costs are amortized over some releases.

b) Version specific test case prioritization:

The test objective is to test the modified version of program. Hence the test case prioritization logic is to determine an ordering of test cases which could be applicable to a specific version of the program. The cost of test case ordering is higher as the specific version of the program needs to be available to verify the test case ordering.

2.1.2 Test case prioritization techniques

Research on test case prioritization technique has been going on since 1989. Roongruangsuwan describes the following techniques for test case prioritization [6]:

a) Customer Requirement-based techniques: In this technique the test cases are prioritized based on the software requirements. Weight factors like requirement volatility, requirement complexity, custom-priority are used to determine priority of the test cases

b) Code coverage based techniques: In this technique the test cases are prioritized based on the maximum code coverage to be achieved by test case. Code coverage

analysis describes the amount of source code executed by the test cases during test execution. The code coverage metric is an indication of software quality [11]. It is a white box technique as direct access to code is required to instrument the code and determine the statements covered by the test case.

The code coverage criteria can be statement coverage, branch coverage, additional branch and additional statement coverage. The test cases are assigned priorities based on the target code coverage percentages. The higher the target code coverage, higher the priority of the test case. The test cases are then ordered in descending order of the priority value.

c) Cost Effectiveness based techniques: In this technique the test cases are prioritized based on cost of test results analysis and cost of test execution. The test cases are assigned priorities based on the cost to execute test cases and defects discovered during previous test runs.

d) Chronographic history based technique: Prioritization of the test cases using this technique is based on the history of data collected from test executions. The criteria can be maximum defect detection, failure rate or test complexity.

Rothermelv[2] lists the following 7 techniques for test case prioritization to achieve high fault detection:

a) Optimal prioritization: The technique is based on exposing faults in the program using prior knowledge of which test cases found which type of faults and prior knowledge of the faults in the program.

b) Total Statement coverage prioritization: In this technique the test cases are ordered based on the highest statements coverage achieved by the tests cases under construction. The test cases are ordered using the Greedy algorithm [2].

c) Additional statement coverage prioritization: In this technique the test cases are ordered based on the highest statements coverage achieved by the tests case and additional statements that were not covered initially. The test cases are ordered using the Greedy algorithm [2].

d) Total Branch coverage prioritization: In this technique the test cases are ordered based on the highest number of branches coverage achieved by the tests case. The test cases are ordered using the Greedy algorithm [2].

e) Additional Branch coverage prioritization: In this technique the test cases are ordered based on the highest number of branches coverage achieved by the test cases and additional branches that were not covered initially. The test cases are ordered using the Greedy algorithm [2].

f) Total fault exposing potential prioritization: This technique is based on the ability of test to find defects in the program. These defects are seeded into the program [2]. Mutation analysis is used to determine the ability of a test case to detect a seeded defect. Test case fault exposing potential is determined based on total faults seeded and total faults detected from the test case.

g) Additional fault exposing potential prioritization: This technique is based on the ability of the tests to find defects in the program where defects are seeded into the program. Mutation analysis is used to determine the ability of a test case to detect

a seeded defect. Test case fault exposing potential is determined based on total faults seeded and total faults detected from the test case.

Another technique suggested by Praveen Rajan Srivastava [3] is based on cost effective test case prioritization. In this technique test cases are prioritized using a priority based scheme. Each test case is assigned a priority based on the higher code coverage, number of faults detected or rate at which faults are detected. The algorithm calculates the average number of faults detected per minute by dividing the number of faults detected by each test by time taken to detect faults. Average number of faults detected per minute is used to determine the priority of test cases in test suite.

Siripong Roongruangsuwan, Jirapun Daengdej [6] proposed a technique called MTSSP to prioritize test cases with same priority across multiple test cases in the same test suite. In this technique test cases are prioritized based on multiple factors as listed below:

a) Defect rate
b) Cost of test execution
c) Time required for test execution
d) Other factors like requirements, test case dependency, test case complexity and test impact.

Test cases are first prioritized by defect rate. If multiple test cases have same priority then it is resolved on the basis of time for test execution. Then other factors like cost, requirements, test case dependency, test case complexity and test impact are considered

till test cases with same priority are resolved. If there are test cases with same priority the test case are prioritized in random order

2.2 Tool analysis

The parameters considered for test tool selection are as described below:

- The tool should support test levels of:
  - Integration testing
  - System testing
- The tool should support different user interfaces such as:
  - IDE Plug-ins, such as Eclipse and Netbeans
  - Command line
  - ANT task
- The tool should support testing of systems developed in different languages
- The tool should generate report in XML
- The tool should support test script development n XML

The below listed testing tools have been analyzed:

a) iValidator [5]: This is an open source automation testing tool suitable for integration, white box and black box testing. The tool supports test script development, test execution and test execution reporting.

b) Jamelon [25]: This is an automation testing framework designed with a plug-in model. The framework supports for 5 plug-ins. The framework has been designed to support integration, functional, acceptance and regression testing.

c) Abbot [26]: This is an automated Graphical User Interface (GUI) testing framework with Junit extension. The framework provides capability to develop unit tests.

Based on the parameters a comparative analysis between three tools namely iValidator, Jamelon and Abbot is illustrated below [5][25][26]:

Parameters	iValidator	Jamelon	Abbot
Unit testing	Yes	Yes	Yes
Integration testing	Yes	Yes	No
System testing	Yes	Yes	Yes
IDE Plug-In (Eclipse)	Yes	Yes	No
Command line	Yes	No	No
ANT task	Yes	No	No
Support testing on systems developed in any language	Yes	Yes	No (only Java)
Generate report in XML	Yes	Yes	No
Test Scripting development in XML	Yes	No (Jelly)	Yes
Cost	Free	Free	Free

Table 2.2-1: Test Tool Comparative Analysis

Abbot has been developed to test Java GUI and hence the tool has lots of features to support GUI testing. As the thesis requirement is not limited to GUI testing, Abbot was not selected. Usability wise iValidator was easier than Jamelon because of the need to write test script in Jelly in Jamelon. iValidator has been developed for integration testing but it can used for unit and system testing as well. As iValidator met most of the requirements and is easy to use, it is preferred over other tools.

2.3 iValidator testing tool

iValidator testing tool has been developed by InfoDesign OSD GmbH. This is an open source testing tool for integration, white box and black box testing. The original idea of developing the tool was to enable testers to develop and execute integration tests. It is based on Java and XML. XML is used to describe the test scripts and test execution results. Tests can be run on different platforms without any modification to the test scripts [5].

2.3.1 Terms and definitions

a) Test step: This term refers to a small sequence of activities between components or within a component in the SuT. For example it indicates a small flow within a use case. Test step needs to be implemented in Java.

b) Test description: This term refers to a test scenario that describes a sequence of activities in a single use case, or a combination of sequences from multiple use cases. A test description is implemented by combining single or multiple test steps. A test step can be used in different test descriptions. Test description is described in XML. Appendix A describes the XML schema for creating test descriptions.

c) Test Suite: This term refers to testing a system level use case. It elaborates sequence of activities that indicate a flow across multiple components in the SuT. Test descriptions are combined to form a test suite. It is described in XML.

d) Test Run: This term refers to the execution of test description or test suite.

2.3.2 Test structure

The Figure 2.3.2-1 represents the test structure in iValidator:

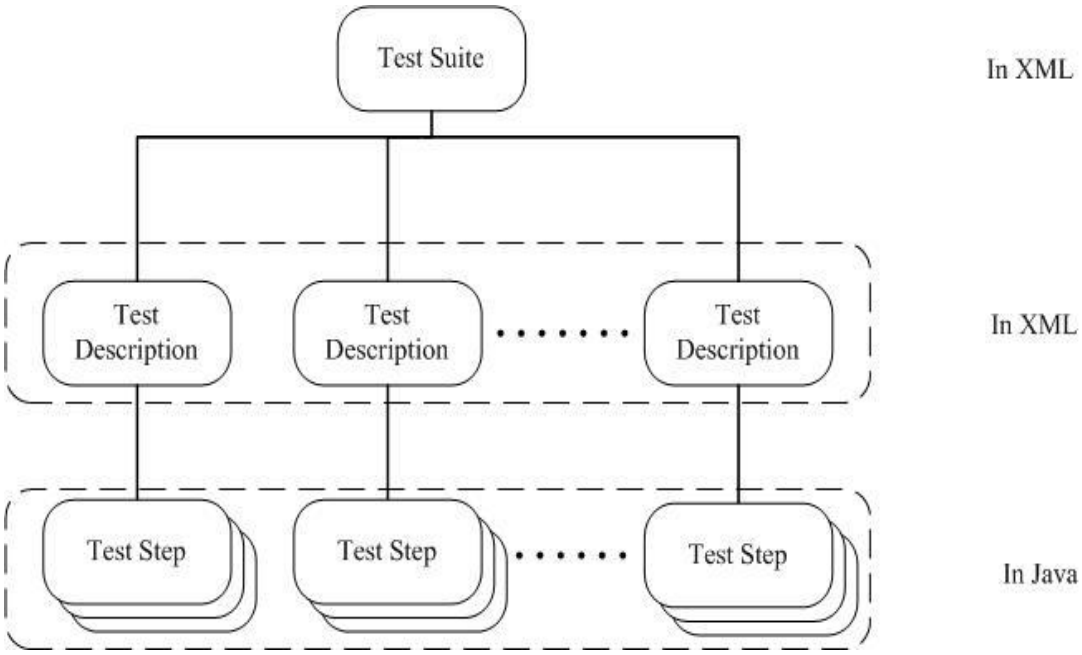

Figure 2.3.2-1 iValidator test structure

The test script development in iValidator is a two step process. The first step is development of test steps. The second step is to write test descriptions using test steps. A test suite creation is optional.

A test run requires the presence of a configuration file. This file contains information about the descriptor repository path, the class path for the SuT and the directory path for storing reports. It is written in XML (Refer Appendix D).

2.3.3 Architecture

The architecture of iValidator is illustrated in Figure 2.3.3-1 below:

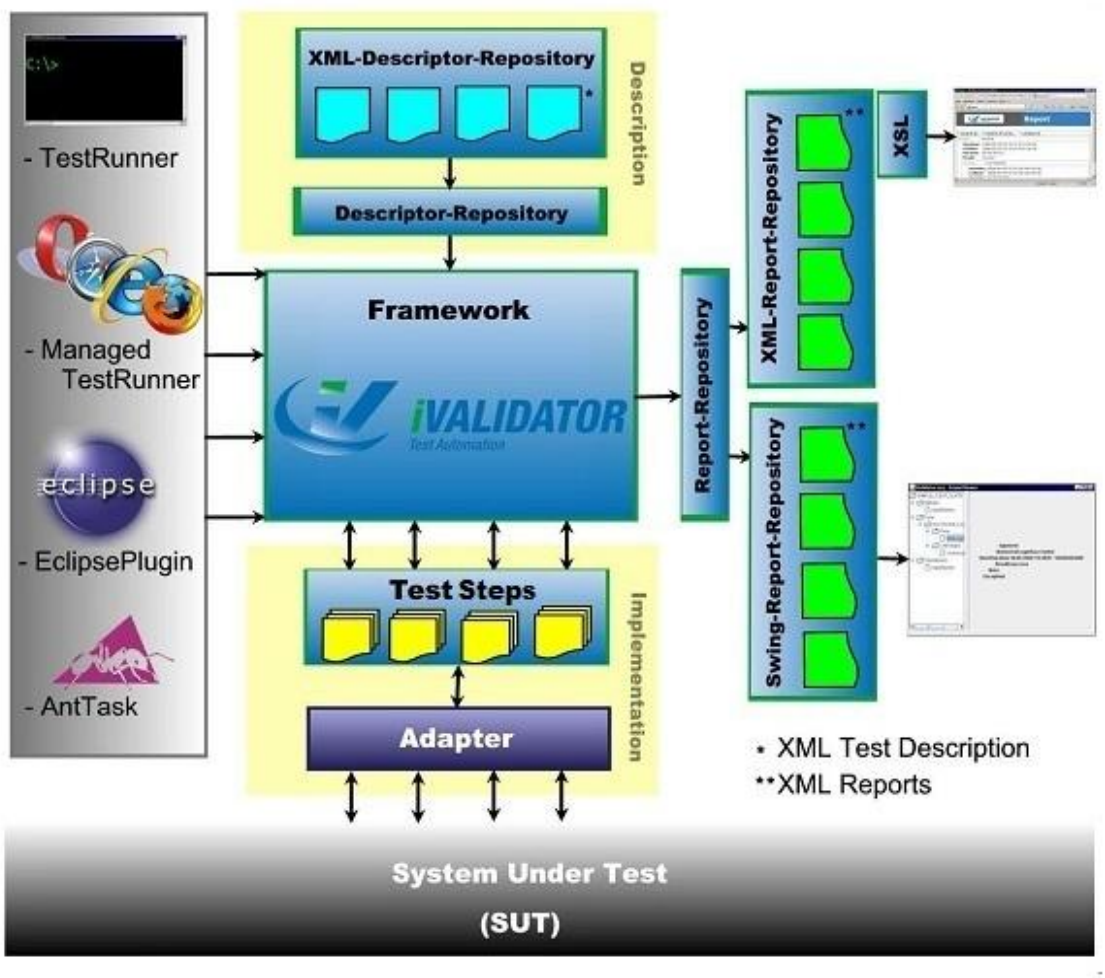

Figure 2.3.3-1: iValidator Architecture [5]

Test execution using iValidator can be scheduled from any of the sources listed below:

a) Command line using iValidator TestRunner

b) Web browser using iValidator ManagedTestRunner

c) Ant task which can be scheduled to execute tests along with the build process

d) Eclipse plug-in

The test execution engine receives test description details from the Descriptor-Repository. This repository contains the names of all the test description files, list of paths to locate test descriptor files and the list of paths to locate test classes in the SuT.

Adapter is an interface between the test steps and the SuT. Test steps contain calls to the methods in the SuT. Hence any modification to the methods in SuT requires modification to the test steps. To prevent the frequent modification of the test steps, iValidator framework provides capability to define an Adapter. It hides the details of the underlying system from the test step implementation.

Test results are generated after the test execution is complete. The result of test run is stored in a repository called Report-Repository. The test execution result indicates test failure or success at both the test description and test step level. The test execution result is stored in XML (Refer Appendix C). The results are displayed in iValidator console in Eclipse. Swing-Report-Repository displays the results on the Graphical User Interface (GUI).

iValidator framework represents a collection of libraries required for test execution and test result reporting. It integrates Descriptor-Repository and Report-Repository.

2.3.4 Class diagram

The iValidator class diagram of classes responsible for test execution is illustrated in Figure 2.3.4-1:

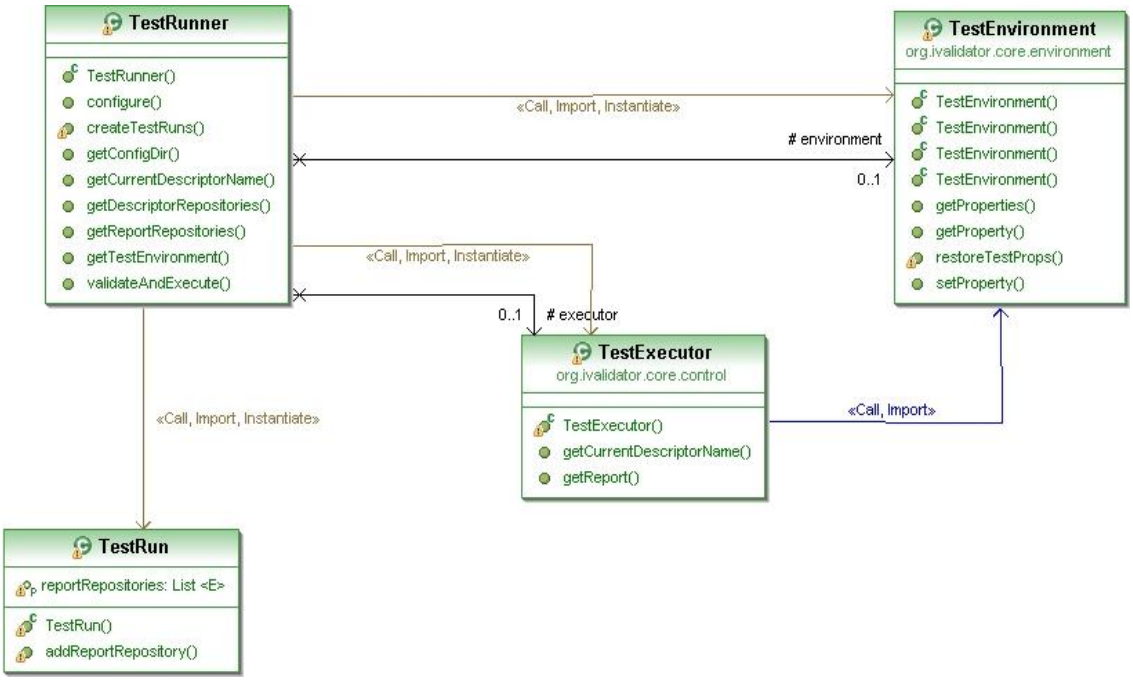

Figure 2.3.4-1 iValidator class diagram

The iValidator classes responsible for test execution are described below:

a) **TestRunner:** This class is instantiated when the test description is selected for execution. The test description details are extracted from the descriptor repositories and the results are stored in the report repository. This class reports the test execution result.

b) **TestRun:** This class contains details about the test steps within a single test description. This class passes the list of test steps to the TestExecutor.

c) **TestExecutor:** This class executes all the test steps mentioned in the test description. The execution of tests is monitored till completion. This class performs test control, which is the monitoring of test execution from the beginning, ability to pause, stop and return the result to the caller once the execution completes. This class is also responsible for creating and destroying adapters.

d) **TestEnvironment:** This class stores the properties required to perform test execution using iValidator. Properties such as the path for iValidator home directory, iValidator library and configuration file are stored in this class. If these properties are not set by tester, default values are assigned.

2.3.5 Features

The primary features of iValidator are listed below:

a) Complex test descriptions: The XML based test descriptions enable the tester to describe test scenarios in a hierarchical format, allowing test descriptions that provide a structured flow using single or multiple test steps. It allows describing a complex sequence of flow of events across multiple components in the SuT, which improves understanding.

b) Supports white box, integration and system level tests.

c) XML based test reports: The results of test execution are stored in XML format. This helps in presenting the test results in multiple formats using XSL and GUI.

d) Reusability of tests: Same test steps can be used in different test scenarios by providing different input data. Re-use of existing test steps helps reduce the effort needed for creating test descriptions.

e) The interfaces available for utilizing the services of iValidator are as listed below:

   i. Eclipse plug-in
   ii. Ant Task
   iii. Command line
   iv. Web browser

f) Adapters wrap access to the SuT. The use of adapters provides a single interface to connect the test with the SuT, thus reducing the effort of maintaining the tests to map to the modifications in the SuT.

g) Parameters for test steps are defined in the XML test descriptor. The test setup initialization and de-initialization can be described as well in the XML descriptor.

h) Logs are provided for each test step in the test scenario

2.3.6 Example

The SuT described in this example is an Order and Stock Management system. The different activities that can be performed using this system are:

a) Customer places an order

b) An order reference number is presented to the customer

c) Customer initiates payment using order reference number

d) Bank makes payment to the system

e) After payment is complete, order and stock management system creates a shipment order for stock manager

A test scenario describing a standard ordering process is illustrated in Figure 2.3.6-1. It describes the sequence of flow of activities between user and different system components. This scenario is described in a test description in XML. The test data "Order data" flows between the test steps.

The test scenario consists of test steps such as Place order, Trigger bank transfer, and Read shipment order. Place order is a single sequence of activities. Place order involves the activities listed below:

a) Selecting an item from list

b) Entering item quantity

c) Adding selected items to the shopping cart

d) Displaying cost for the items in the shopping cart

Select goods, Shipment, Bank transfer are adapters, as they interface the test steps with the order and stock management system. The system might need to interact with third party, which in this example is the Bank. A simulator is used that mimics the behavior of the Bank entity.

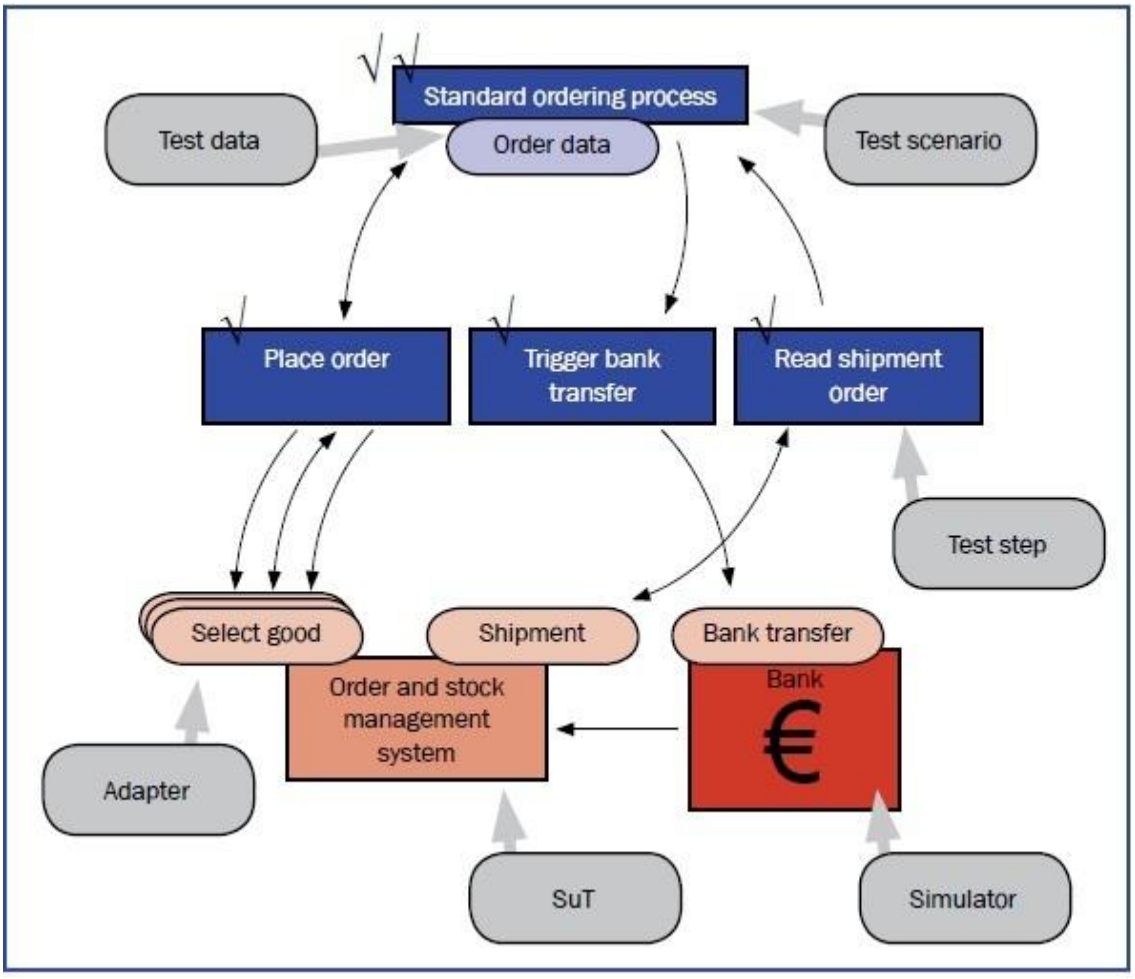

Figure 2.3.6-1: iValidator test example

3 Use case model

The use case diagram of extended iValidator with design and implementation of test step prioritization is illustrated in Figure 3-1 below:

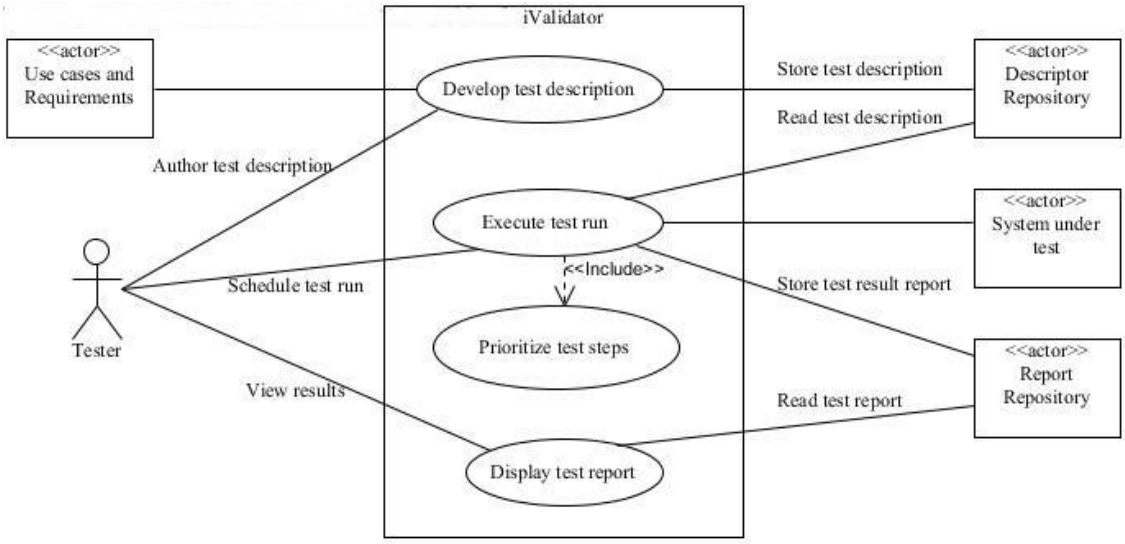

Figure 3-1: Use case model

The use cases are described below:

Use case 1: Develop test description

Actor: Tester

Precondition: Requirements for system under test are established and baselined

Main Scenario:

1	Tester develops test steps based on requirements
2	Tester develops test descriptions using test steps from Step 1
3	Tester updates test description file path in Descriptor-Repository (Refer section 2.3.3)

Use case 2: Execute test run

Actor: Tester

Prerequisite: Test case descriptions have been reviewed and approved

Includes: Prioritize test steps

Main Scenario:

1	Tester schedules execution of test description
2	TestRunner (Refer section 2.3.3) derives the test description details from Descriptor-Repository
3	Priority of the test steps is determined as described in use case #3
4	TestExecuter (Refer section 2.3.3) executes the test steps in the prioritized order on the SuT
5	Test execution results are saved in Report-Repository

Use case 3: Prioritize test steps

Actor: Execute test run

Prerequisite: Test description to be executed are verified and approved

Main Scenario:

1	Derive test execution history for test steps, static analysis and complexity analysis for classes accessed by test steps in SuT
2	Compute the priority of test steps in test description based on the algorithm (Refer to section 5.5)
3	Order test steps in descending order of priority
4	Return the ordered list of test steps

Use case 4: Display test report

Actor: Tester

Prerequisite: Test execution results are available

Main Scenario:

1	Derive test results from Report-Repository (Refer section 2.3.3)
2	Tester views the swing console or XSL console to check the test results

4 Functional requirements

The requirements for extending iValidator test tool to include test step prioritization are described in table 4-1:

Requirement #	Name	Description
F.R.1	iValidator	F.R.1.1 The iValidator framework shall be modified to execute test steps in a test description in prioritized order in accordance with algorithm described in section 5.5 F.R.1.2 Test step priority shall be determined for test steps scheduled for execution F.R.1.3: The test step metric information shall be displayed in web page
F.R.2	Test Step Prioritization	F.R.2.1: Priority shall be calculated for each test step in a test description F.R.2.2: Test execution history shall be derived for every test step in test description F.R.2.3: Complexity analysis shall be derived for the class in SuT accessed by test steps in test description F.R.2.4:Static analysis shall be derived for the class in SuT accessed by test steps in test description F.R.2.5 : Test step priority shall be determined based on test execution history, complexity analysis and defects

		identified from static analysis F.R.2.6: The test execution history, static analysis and complexity analysis of the test steps shall be stored in test step metric repository XML in section 6.6
F.R.3	Static analysis	F.R.3.1: Static analysis shall be performed for classes in SuT accessed by test steps in test description. F.R.3.2: Static analysis results shall have a severity value assigned F.R.3.3: Static analysis shall be able to perform analysis on byte code of SuT F.R.3.4: Static analysis shall be able to perform analysis on source code of SuT F.R.3.5: Static analysis result shall indicate the defect type, line number in class and class name in SuT F.R.3.6: The number of defects identified for each class shall be calculated
F.R.4	Complexity Analysis	F.R.4.1: Complexity analysis result shall be calculated on a per class basis for classes accessed by test step or test steps in test description

Table 4-1: Functional Requirements

5 Design specification

This chapter describes the design for the test step prioritization algorithm. It also discusses the rationale behind the algorithm and includes the pseudo-code for implementing the algorithm.

5.1 Test step prioritization algorithm

The test objective to be achieved by the test step prioritization algorithm is to increase rate of defect detection. *"Rate of defect detection is a measure of how quickly defects are identified in testing process" [23].* In order to increase the rate of defect detection, the order of the test steps can be determined by identifying defect proneness of the code accessed by each test step. The test steps that access higher defective code need to be executed earlier that other test steps. By doing so defects can be identified earlier than when test steps are executed in random or sequential order. To determine the defect prone code in the SuT, the below listed parameters are considered:

a) Static Analysis

b) Complexity Analysis

c) Test execution history

5.2 Test step prioritization algorithm assumptions

Test step prioritization algorithm makes the below listed assumptions:

a) Each test step is an independent thread of execution

b) Each test step can access one or more classes in the SuT

c) Test description will be a collection of independent test steps

d) All identified defects have the same severity

5.3 Test step prioritization algorithm rationale

The rationale for selecting static analysis, complexity analysis and test execution history as parameters to determine test step priority are listed below:

- Test execution history: This parameter helps determine whether a test step has failed or passed, the number of times the test step has been executed and test step execution time. A failed test step indicates the presence of a defect. The failed test step needs to be executed in the next execution cycle in order to verify whether a defect has been fixed. The frequency of failure of a test step is indicative of defect prone code. The frequency of test step failure is determined by the Test step Failure Rate (TFR) metric. This metric indicates the number of times the test step has failed over previous test runs. Using the test step execution data from previous test runs, TFR can be calculated as:

  $$\text{Test step Failure Rate (TFR)} = \frac{\text{Number of times test step has failed}}{\text{Total number of times test step has been executed}}$$

  Higher TFR of a test step indicates the presence of a higher defect prone code when compared to that of a lower TFR of a test step. Hence, such higher TFR test steps need to be executed prior to the test steps that have a lower TFR. In test step prioritization algorithm the TFR is calculated for every test step. iValidator maintains the test execution history of each test step.

- Static analysis: Static analysis of the SuT can reveal certain types of defects without executing the code. Examples of such defects are: null pointer de-references, incorrect string manipulation, and memory leaks. The identified defect(s) is indicative of the presence of a defective code. The number of such defects is an important metric to be considered for test step prioritization. Hence,

a test exhibiting a higher number of defects needs to be tested earlier than other tests with lower number of defects.

- Complexity Analysis: Many metrics are used to indicate code complexity like Lines Of Code (LOC), McCabe's Cyclomatic Complexity and Halstead's complexity measure [11]. McCabe's cyclomatic complexity indicates the complexity based on control flow graph of every module of a program. Complex programs are difficult to maintain and understand. Additionally, complex programs are prone to contain more defects [11][24]. Higher complexity indicates the need for more testing efforts [11][24]. Hence it is an important metric to be considered for test step prioritization as it is indicative of defect proneness of the code in the SuT.

5.4 Test step prioritization algorithm description

The test step prioritization calculates the priority of test steps based on static analysis, complexity analysis and test execution history. If some test steps have the same priority value then the order is determined based on complexity analysis. The basic flow diagram of the algorithm is illustrated in figure 5.4-1:

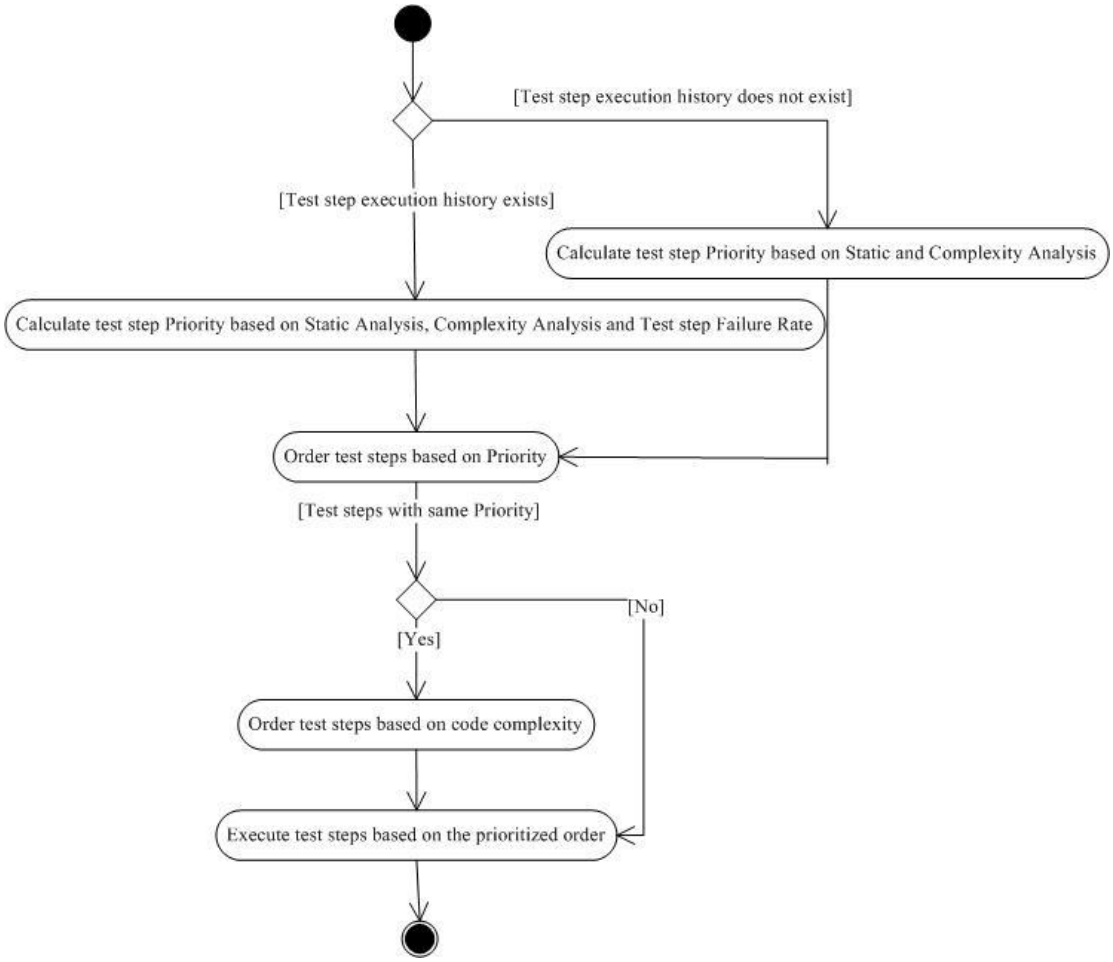

Figure 5.4-1 Flow chart of Test Case Prioritization Algorithm

The process for calculating the Priority of the test steps is illustrated in Figure 5.4-2:

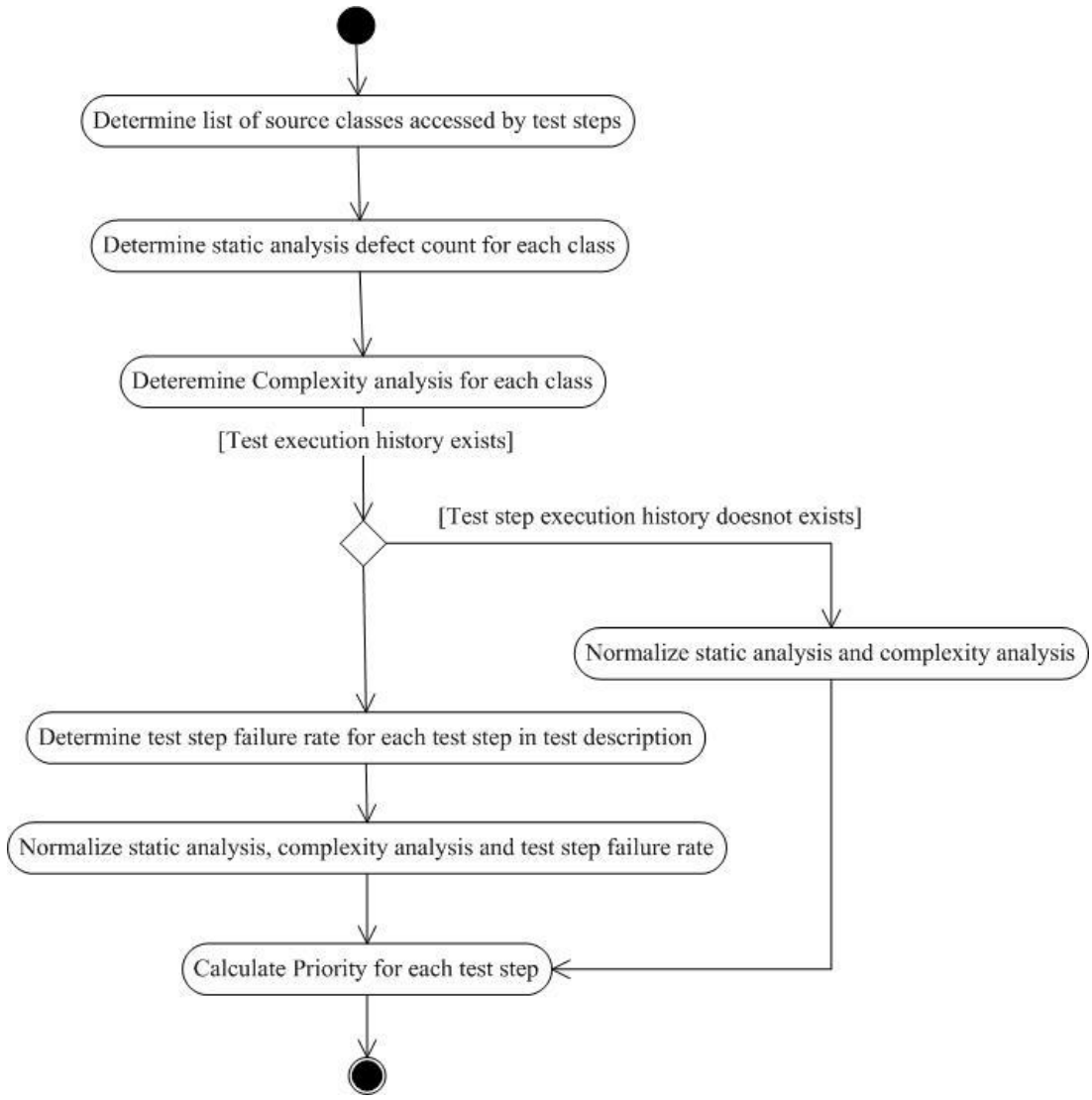

Figure 5.4-2: Process for Calculating priority of test step

The first step in the algorithm is to identify the classes in the SuT being accessed by the test steps under execution. A list of classes being accessed by each test step under execution is created using the Test Step to Source class mapping xml (See section 6.5). The list L will consist of unique classes, i.e., duplicate classes will be considered only once for each test step.

Static analysis is performed on the classes in the list L. This operation fetches the number of defects identified for each class. If a test step accesses at least one class, the average of

static analysis defect count (SC) of all such accessed classes is calculated and assigned to the test step.

McCabe's cyclomatic complexity (CC) is determined for each class in the list L. If a test step accesses at least one class, the average of cyclomatic complexity of all such classes is calculated and assigned to the test step.

If a test execution history exists for the test step, then Test step Failure Rate (TFR) is calculated by dividing the number of times the test step failed by the total number of times test step has been executed.

The values of SC, TFR and CC are in different units of measurement. Calculating priority based on parameters in different units is difficult. Hence it is required to have the three parameters in the same unit of measurement. Normalization expresses all three parameters in a single unit of measurement. Normalization is performed by identifying the contribution of SC or TFR or CC of each test step towards the overall SC or TFR or CC of all the test steps in a test description. Normalization is performed in the below listed steps:

a) Calculate the Total SC of all the test steps in the test description
b) Calculate the Total TFR of all the test steps in the test description
c) Calculate the Total CC of all the test steps in the test description
d) For each test step, calculate the ratio of SC to the Total SC, as defined in step 1
e) For each test step, calculate the ratio of CC to the Total CC as defined in step 3
f) For each test step, calculate the ratio of TFR to the Total TFR as defined in step 2

For each test step Priority is calculated using the formula listed below:

Case 1: If test step execution history exists:

Priority = Normalized TFR + Normalized SC + Normalized CC

Case 2: Test step execution history does not exist:

Priority = Normalized SA + Normalized CC

The test steps in a test run are ordered in the descending order of Priority. If Priority value of any two test steps is the same, then the CC values of the individual test steps are considered to resolve the conflict and determine a new order. If the CC values and the computed Priority values of any two test steps are the same, then the order of the test steps is maintained as is. The rules described above are used repeatedly to resolve any conflict that may exist in the order of the test steps in a test run.

5.5 Test step prioritization algorithm pseudo code

The test step prioritization algorithm described earlier is presented as a pseudo code as illustrated below:

Input:
T: set of test steps "t1…tn"
C: set of Classes "C1…Ck" accessed by test step
Output: A prioritized list of test cases in T

```
1.begin:
2.  For (all t in T)
3.  {
4.       If (Execution history of t exists)
5.            Calculate TFR for t
6.       Else
7.            Initialize TFR to zero for t
8.       SC_t ← Compute Static Analysis Defect for C_t
9.       CC_t ← Compute Cyclomatic Complexity for C_t
10.       }
11.       For (all t in T)
12.       {
13.            NTFR_t = Normalize TFR_t
14.            NSC_t = Normalize SC_t
15.            NCC_t = Normalize CC_t
16.            Priority_t = NTFR_t + NSC_t + NCC_t
17.       }
```

```
18.  Sort(T):
19.   Begin Sort
20.       If Priorityt1 > Priorityt2
21.          Order → t1, t2
22.       Else if Priorityt1 < Priorityt2
23.          Order → t2, t1
24.       Else if CCt1 > CCt2
25.          Order ← t1, t2
26.       Else if CCt1 < CCt2:
27.          Order ← t2, t1
28.       Else
29.          Order ← t1, t2
30.   end Sort
31. end
```

6 Detailed design

This chapter describes the detailed design for the software product developed in this project. Contents include a discussion on the changes made to iValidator architecture, the modified iValidator class diagram and sequence diagrams related to the added functionality. It also discusses the tool selection for static and complexity analysis.

6.1 iValidator modified architecture

The iValidator architecture as discussed in section 2.3.3 is modified to incorporate the test step prioritization functionality as illustrated in Figure 6.1-1:

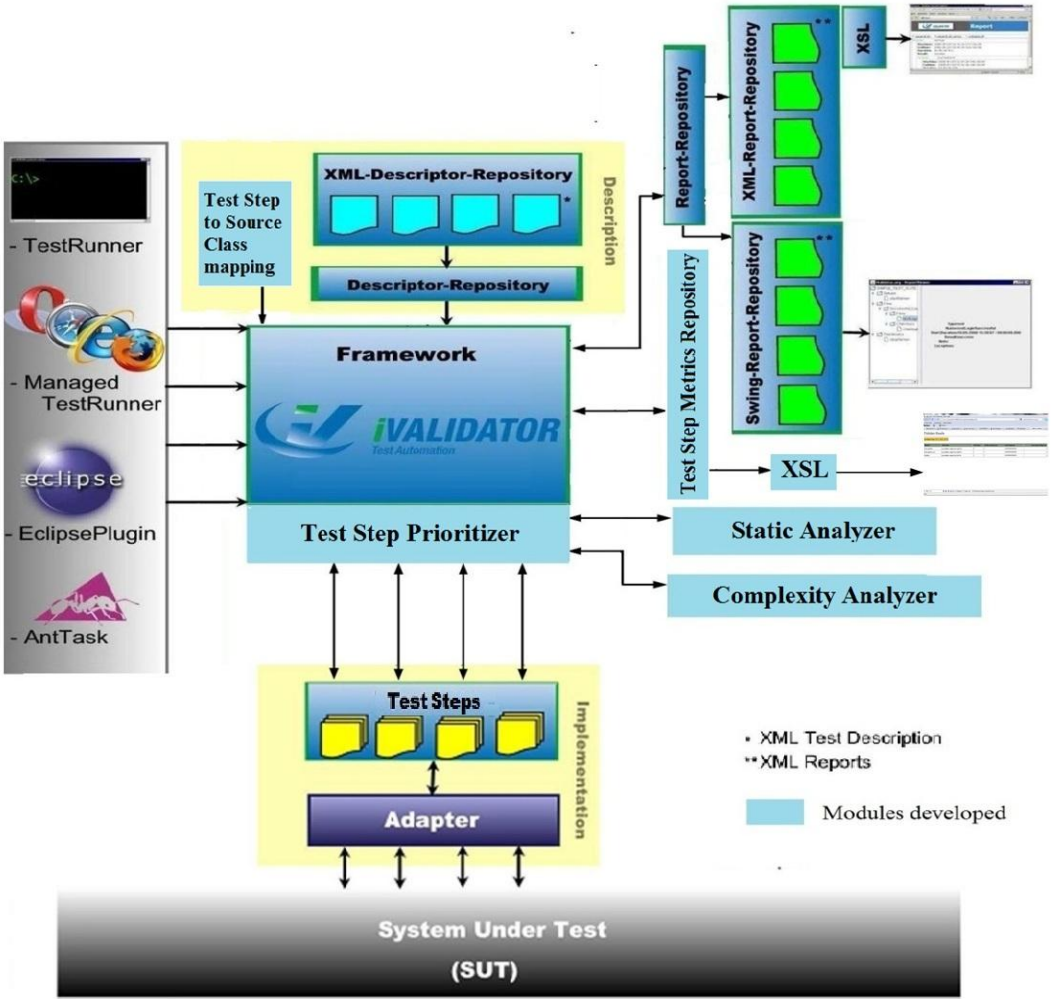

Figure 6.1-1: Modified iValidator Architecture

iValidator architecture has been modified by integrating the below listed modules:

a) Test Step Prioritizer: The objective of test step prioritizer is to calculate Priority for each test step in a test description. It derives the class names accessed by each test step using Test Step to Source class mapping xml (See section 6.5). It interfaces with Static Analyzer and Complexity Analyzer to read values of defect count from static analysis and McCabe's cyclomatic complexity from complexity analysis for each test step. It reads the test step execution history from Test Step Metric Repository. It is invoked by the iValidator framework when the test run is scheduled. It provides the prioritized list of test steps for execution. It saves the defect count from static analysis, McCabe's cyclomatic complexity from complexity analysis for the test steps in Test Step Metrics Repository.

b) Static Analyzer: The objective of this module is to perform static analysis on the classes being executed by the test steps. It is invoked by Test Step Prioritizer module. The output from this module is the list of static analysis defects for each class. The output is returned to Test Step Prioritizer.

c) Complexity Analyzer: The objective of this module is to determine McCabe's cyclomatic complexity for the classes being executed by the test steps. It is invoked by Test Step Prioritizer module. The output is McCabe's cyclomatic complexity for each class. The output is returned to Test Step Prioritizer.

d) Test Step Metrics Repository: It is a repository of data collected for every executed test step. The test step failure rate, complexity analysis and static analysis defect count are considered as test step metric. The Test Step Metric Repository is stored in XML (See section 6.6).

6.2 Class diagram

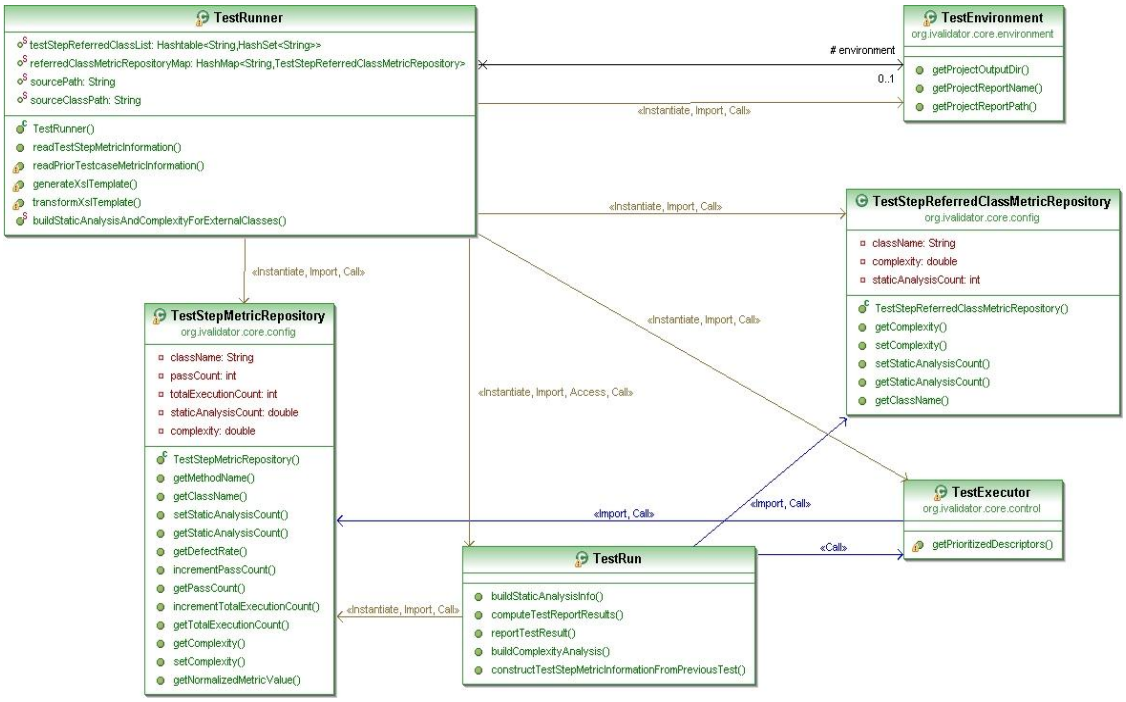

Figure 6.2-1: Modified iValidator class diagram

The original class diagram in section 2.3.4 has been modified and illustrated in Figure 6.2-1. It represents the modified iValidator class diagram to add the test step prioritization algorithm. The existing classes of TestRunner, TestExecuter, TestRun and TestEnvironment have been modified to provide additional functionality. New classes of TestStepReferredClassMetricRepository and TestStepMetricRepository have been created. The functionality provided by the classes is as described below:

1) TestEnvironment: The description is the same as section 2.3.4.
2) TestRunner: The test step metric information is loaded from the test step metric repository xml file. The test step to class mapping information is loaded from the test step to source class mapping xml file. It builds the static analysis and complexity analysis for classes accessed by all the test steps.

3) Test Executor: This module is responsible for calculating the priority of test steps and executing them in the prioritized order on the SuT.
4) TestRun: This module sets the McCabe's cyclomatic complexity and static analysis defect count on each test step. TestStepMetricRepository objects corresponding to the test steps are updated with the metric values of current execution. This module passes the list of all the test steps to the TestExecutor module.
5) TestStepMetricRepository: This class maintains the test step metric information for a test step in the test description. It stores the test step pass count, test step execution count, static analysis and complexity analysis for the test step.
6) TestStepReferredClassMetricRepository: It maintains the static and complexity analysis values for all the classes under current test execution.

6.3 Sequence diagram

The interaction between classes is illustrated in the sequence diagram in Figure 6.3-1:

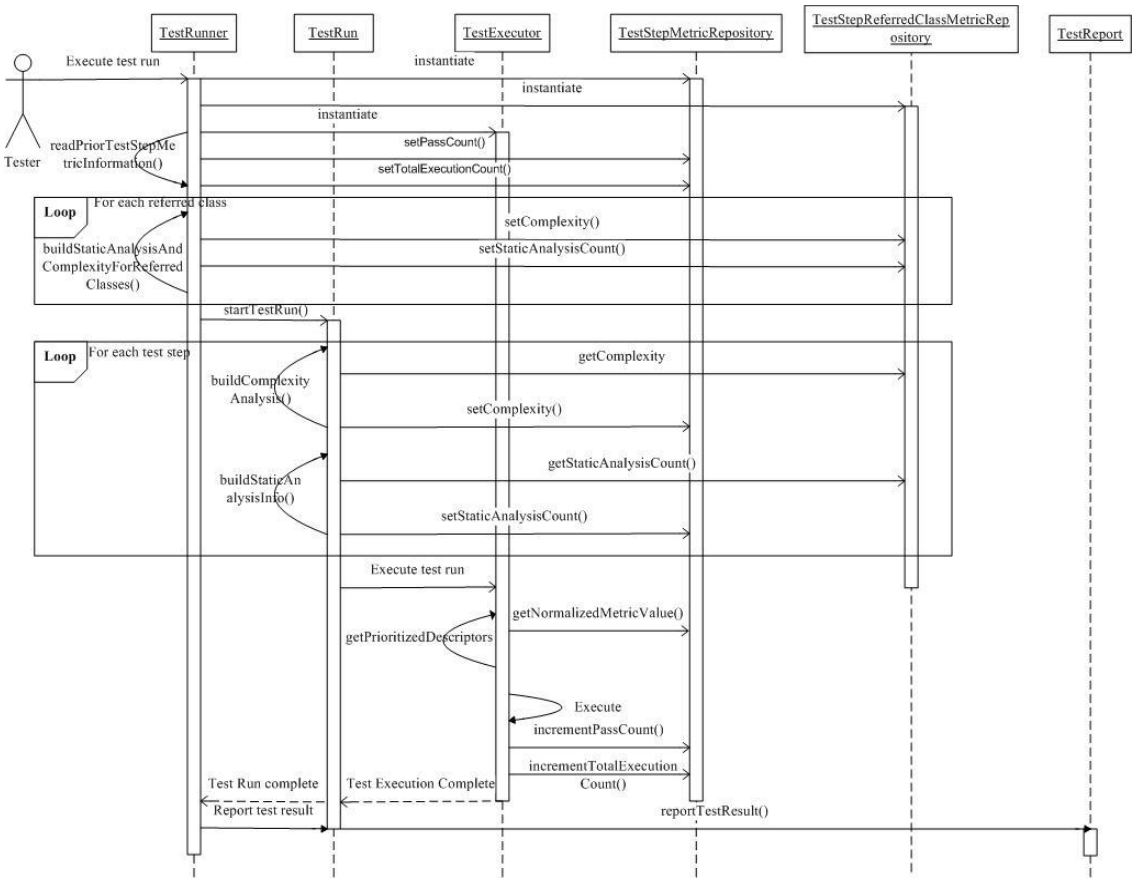

Figure: 6.3-1: Sequence diagram for class interaction

The following sequence is executed in a test run:

1) TestRunner loads the test step metric repository file when a test description is selected for execution. The pass count and total execution count for each test step is set in the corresponding TestStepMetricRepository object.
2) TestRunner loads the mapping between test steps and classes under test (referred class) from Test Step to Source class mapping file.
3) TestRunner performs static analysis and complexity analysis on the referred classes and stores the information in TestStepReferredClassMetricRepository objects.
4) TestRunner initiates the execution of the test description

5) TestRun fetches the static and complexity analysis of the referred classes created in step 3 for each test step. This information is stored in TestStepMetricRepository object of each test step.
6) TestRun passes the control to TestExecutor to start test execution
7) TestExecutor calculates the priority of the test steps based on the normalized test step metric derived from TestStepMetricRepository object for each test step.
8) TestExecutor sorts the test steps in the descending order of priority
9) TestExecutor orders the test steps with same priority based on complexity value of the test steps
10) TestExecutor increments the pass count upon successful execution and increments the total execution count in TestStepMetricRepository object for each test step
11) TestExecutor notifies TestRun that the test execution is complete
12) TestRun notifies TestRunner that test execution is complete
13) TestRunner notifies TestReport to report the test execution results
14) TestReport uses the TestStepMetricRepository objects to generate the reports for the execution of the test steps in the test description

6.4 Tools for test step prioritization

Static and complexity analysis functionality in iValidator have been provided using open source code tools. This section describes the tools that were considered as candidates for integration into iValidator.

6.4.1 Complexity analysis tool selection

To enable complexity analysis in iValidator, two candidate tools namely JavaNCSS and Cobertura were considered. A brief description of the tools is as described below:

- JavaNCSS [28] is a command line utility which calculates McCabe's cyclomatic complexity and Non Commenting Source Statements (NCSS) for methods, classes and packages. The output is generated in XML.
- Cobertura is a test code coverage tool [16]. It determines the parts of the Java source code accessed by tests during execution. It calculates the code coverage in terms of the percentage of lines and branches covered for every class. It calculates McCabe's cyclomatic complexity for each class, package and overall project. It generates report in XML and HTML.

Cobertura and JavaNCSS meet the requirements for complexity analysis mentioned in F.R.4 in section 4. Cobertura has been selected for integration into iValidator. The reasons for selecting

Cobertura over JavaNCSS are as listed below:

a) Cobertura supports Generics in Java, which is not supported JavaNCSS
b) Cobertura supports Annotation in Java, which is not supported by JavaNCSS
c) Cobertura development activities are in progress and patches are released for the reported bugs. However development and bug fixes to JavaNCSS have ceased since 2009

6.4.2 Static analysis tool selection

The requirements for static analysis tool are:

- Result: The results should be stored in XML.

- Built-in and Custom rules: Rule defines a check to be performed during code analysis. For example a rule can be a check to avoid nesting multiple classes. The tool must have a set of predefined rules for the language concerned. An example of predefined rule is Double assignment of a local variable: Assigning a value twice to the local variable might be a logical error or a typing mistake. There should be a capability to develop custom rules.
- Source code and byte code analysis: The tool should be capable of performing static analysis on byte code and source code
- Severity for defects: Severity indicates the level of deviation from the rule. The tool should indicate the severity for the defects identified. Severity can be High, Low and Medium.

The below listed static analysis tools were analyzed:

1) Findbugs [15]: It is used to identify defects such as null pointer exception and memory leak in Java code. It has built-in rules to check the Java code for defects.
2) PMD [14]: It analyzes Java code to determine defects such as complicated expressions, dead code, empty if else and duplicate code. It allows custom rules and has pre-defined built-in rules to check the code for defects.
3) Checkstyle [13]: It can be used for checking source code for compliance to certain coding rules in Java. It checks for duplicate code, naming conventions and others. A configuration file with details on the checks to be performed on the source code is required.

6.4.1.1 Comparative analysis for static analysis tools

The criteria for comparative study of the static code analysis tools is based on the requirements mentioned in section 6.4.1. The comparative study of features of each tool is illustrated in table 6.4.1.1 -1 [13] [14] [15]:

Static Code Analysis Tools	Results in XML	Defining Rule	Byte code and Source Code of system under test	Severity assigned for defects	Source code environment
Findbugs	Yes	Built-in and Custom	Source and Byte code	Yes	Java
PMD	Yes	Built-in rules and Custom	Source code	No	Java
CheckStyle	No	No	Source code	Yes	Java

Table 6.4.1.1-1: Comparative analysis of open source static code analysis tools

Based on the comparative analysis Findbugs fulfills most of the requirements for the static code analysis. Hence Findbugs has been selected to be integrated into iValidator to perform static code analysis.

6.5 XML schema for test step to source class mapping

Static and complexity analysis have to be performed on the classes in the SuT being accessed by the test steps under execution. The XML schema defines a format for providing the details about the classes in the SuT being accessed by each test step. This information is used to perform static and complexity analysis on the source classes. Tester needs to create this file and keep the information up to date for each test step.

The test step to source class mapping file is required because of the difficulty in deriving the details of the SuT classes accessed by the test steps. The major difficulty lies in making significant changes to the iValidator execution engine to include Byte Code Engineering Library (BCEL). The XML schema for Test Step to Source class mapping is illustrated below:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="sourceClassMapping" type="sourceClassMapping"/>
 <xsd:complexType name="sourceClassMapping">
  <xsd:sequence minOccurs="1" maxOccurs="1">
   <xsd:element name="sourcePath" type="xsd:string""/>
   <xsd:element name="sourceClassPath" type="xsd:integer"/>
   <xsd:element name="teststeps" type="teststeps"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="teststeps">
  <xsd:choice minOccurs="1" maxOccurs="unbounded">
   <xsd:element name="teststep" type="teststep" />
  </xsd:choice>
 </xsd:complexType>
 <xsd:complexType name="teststep">
  <xsd:choice minOccurs="1" maxOccurs="1">
   <xsd:element   name="referringSourceClasses"   type="referringSourceClasses

  </xsd:choice>
 </xsd:complexType>
 <xsd:complexType name="referringSourceClasses">
  <xsd:choice maxOccurs="unbounded">
   <xsd:element name="referringSourceClass" type="referringSourceClass" />
  </xsd:choice>
 </xsd:complexType>
 <xsd:complexType name="referringSourceClass">
  <xsd:attribute name="name" type="xsd:string" use="required"/>
 </xsd:complexType>
</xsd:schema>
```

An example of the test step to source class mapping file is illustrated below:

```
<?xml version="1.0" encoding="UTF-8"?>
<sourceClassMapping>
   <sourcePath name="../org.ivalidator.sample.testapp/source"/>
   <sourceClassPath name="../org.ivalidator.sample.testapp/classes"/>
   <teststeps>
      <teststep class="org.ivalidator.sample.base.LoginTest"
method="testLoginSuccessful">
         <referringSourceClasses>
            <referringSourceClass
name="org.ivalidator.sample.testapp.ServerImpl"/>
            <referringSourceClass
name="org.ivalidator.sample.testapp.AccessDeniedException"/>
         </referringSourceClasses>
      </teststep>
      <teststep class="org.ivalidator.sample.base.LoginTest"
method="testLoginFails">
         <referringSourceClasses>
            <referringSourceClass
name="org.ivalidator.sample.testapp.ServerImpl"/>
         </referringSourceClasses>
      </teststep>
   </teststeps>
</sourceClassMapping>
```

The source classes specified in the test step to source class mapping xml file are used by Static analyzer (See section 6.1) and Complexity analyzer (See section 6.1) to perform static and complexity analysis respectively. The information conveyed by the above example is as described below:

1) Test step representing method testLoginSuccessful in class org.ivalidator.sample.base.LoginTest accesses the below listed classes in SuT:
    - org.ivalidator.sample.testapp.ServerImpl
    - org.ivalidator.sample.testapp.AccessDeniedException
2) Test step representing method testLoginFails in class org.ivalidator.sample.base.LoginTest accesses the below listed class in SuT:
    - org.ivalidator.sample.testapp.ServerImpl

6.6 XML schema for test step metric repository

Test step metric repository maintains test execution result data, static and complexity analysis results for each test step. This repository is updated with the results of all the test steps after every test run. The repository maintains information for test steps executed in earlier test runs. The information maintained in the repository is:

1) Test step pass count: Number of times test step has passed
2) Test step total execution count: Number of times test step has been executed
3) Cyclomatic Complexity: McCabe's Cyclomatic Complexity
4) Static analysis defect count

The XML schema for Test Step Metric Repository is described below:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="IValidatorResults" type="IValidatorResults"/>
 <xsd:complexType name="IValidatorResults">
  <xsd:choice minOccurs="1" maxOccurs="unbounded">
   <xsd:element name="TestDescription" type="TestDescription"/>
  </xsd:choice>
 </xsd:complexType>
 <xsd:complexType name="TestDescription">
  <xsd:choice minOccurs="1" maxOccurs="1">
   <xsd:element name="TestSteps" type="TestSteps" />
  </xsd:choice>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
 </xsd:complexType>
 <xsd:complexType name="TestSteps">
  <xsd:choice minOccurs="1" maxOccurs="unbounded">
   <xsd:element name="TestStep" type="TestStep" />
  </xsd:choice>
 </xsd:complexType>
 <xsd:complexType name="Teststep">
  <xsd:sequence minOccurs="1" maxOccurs="1">
                <xsd:element name="PassCount" type="xsd:integer""/>
                <xsd:element name="TotalExecutionCount"
            type="xsd:integer"/>
                <xsd:element name="CyclomaticComplexity"
            type="xsd:double"/>
                <xsd:element name="StaticAnalysisDefectCount"
            type="xsd:integer"/>
              </xsd:sequence>
              <xsd:attribute name="name" type="xsd:string" use="required"/>
              <xsd:attribute name="class" type="xsd:string" use="required"/>
             </xsd:complexType>
            </xsd:schema>
```

7 Implementation

The test step prioritization algorithm has been implemented in Java using Eclipse environment. The algorithm implementation and changes to iValidator were performed in phases as described below:

- Understanding iValidator source code: A tool called eUML2 free edition has been utilized to create class and sequence diagrams in Eclipse. This tool was used to recover class design through reverse engineering of the code. The tool is capable of discovering class properties such as attributes, dependencies and class associations in the iValidator source code.

  Executing iValidator test runs in debug mode within Eclipse has helped in understanding the flow of data between and within classes.

- Integrating Findbugs and Cobertura into iValidator: Based on the study of the iValidator source code, the TestRunner class was modified to incorporate calls to Findbugs and Cobertura APIs. The APIs for both the tools were modified to perform analysis on the on a specific input list of classes. The jar files of both the tools are bundled together with the iValidator jar file.

- Implementing test case prioritization algorithm: The TestExecutor class was modified to calculate priority for each test step as defined in the algorithm in section 5.5.

- Implementing XSL Transformation to display test metric report: The TestRunner class was modified to perform XSL Transformation of the Test Step Metric Repository xml file. The results can be viewed in an internet browser.

To track changes and to maintain revision control, the Eclipse workspace was maintained with Perforce [19]. The free edition of Perforce was downloaded and installed. The iValidator source workspace was added to Perforce and thus all the changes made during implementation were under version control.

8 Testing

The test case prioritization algorithm was tested using the test scenarios described below:

a) Test description with one test step

b) Test description with multiple test steps

c) Test description with test steps using methods from one class

d) Test description with test steps using methods from different classes

e) Test description with test steps having same priority

A sample Java application was selected to act as a SuT. Test setup was created with test step, test descriptions, test step to source class mapping and configuration file. The test setup consists of test descriptions based on the five scenarios mentioned above. The test steps were implemented in Java to test the SuT. At the end of each test execution a Test Step Metric Repository XML file and test execution result XML file were created. The result of the test execution is displayed in iValidator console as illustrated below:

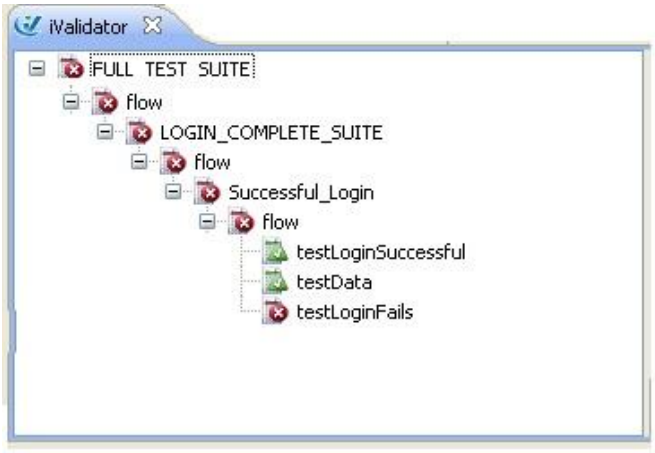

Figure 8-1: iValidator test result console

The test step metric repository information is displayed in HTML. The result is displayed in a tabular format listing the name of the test description and test steps. Each test step in a test description records a pass count, total execution count, McCabe's complexity, and

static analysis defect count. The test step metric repository information is displayed as illustrated below.

IValidator Results

Test Name	TestUserLogin

Test Step	Test Step Class Name	PassCount	TotalExecutionCount	CodeComplexity	StaticAnalysisBugCount
testLoginFails	org.ivalidator.sample.base.LoginTest	0	3	2.090909090909091	4
testLoginSuccessful	org.ivalidator.sample.base.LoginTest	1	3	2.090909090909091	1
testData	org.ivalidator.sample.base.DataTest	1	1	1.090909090909091	2

Figure 8-2: Test step metric repository

In Figure 8-2, testLoginFails is a test step in class org.ivalidator.sample.base.LoginTest. For this test step, the pass count is 0, which indicates that the test step never passed. Total execution count, complexity and static analysis defect count are also reported. Similar information is displayed for test steps testLoginSuccessful and testData.

9 Conclusion

This chapter concludes the thesis for the Design and Implementation of test step prioritization in iValidator. Included within this chapter are a summary of the goals and objectives achieved, benefits and limitations, and some thoughts regarding recommendations for future research.

9.1 Summary of goals and objectives

The objectives of designing test step prioritization algorithm and implementation of the algorithm in iValidator have been successfully achieved. iValidator has been successfully modified to prioritize test steps in a test description. Test execution history, static and complexity analysis have been accounted towards the calculation of test step priority. Findbugs and Cobertura have been successfully integrated into iValidator to perform static and complexity analysis respectively.

9.2 Benefits and Limitations

The benefits of test step prioritization algorithm in iValidator are as described below:

a) The algorithm can be used to order test steps in regression and non-regression testing. As the algorithm is designed to calculate priority in the absence of test execution history, it can be used in non-regression cycle. In regression testing cycle, the test execution history is available and can be used to calculate priority.

b) The test step priority is determined based on parameters that are affected by changes to source code. Hence it can be used on any version of the SuT.

Some limitations in the extended iValidator tool are as described below:

a) Findbugs and Cobertura perform static and complexity analysis for Java based applications only. This limits the use of iValidator tool to Java based software.
b) The test step prioritization algorithm assumes that the test description would not indicate a test scenario and would be a collection of independent test steps. This assumption does not allow for the specification of a specific flow of test steps in the test description.
c) The test step prioritization assumes all defects have same severity.

9.3 Future work and recommendations

The test step prioritization algorithm can be extended to prioritize test descriptions. To achieve this objective, a weighted formula can be designed to calculate the priority of test description based on the priority for each test step.

Static and complexity analysis tools that perform analysis on other languages can be integrated into iValidator. This will be helpful in performing test step prioritization for SuT developed in languages other than Java.

The test step prioritization algorithm can be extended to consider severity of defects, so that test steps detecting higher severity defects are ordered before other test steps.

References

1) Sebastian Elbaum Alexey Malishevsky Gregg Rothermel, 2001. Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. IEEE ICSE '01 Proceedings of the 23rd International Conference on Software Engineering, ISBN:0-7695-1050-7
2) Gregg Rothermel,Roland H. Untch, Chengyun Chu, 2001. Prioritizing Test Cases For Regression Testing. IEEE Transactions on Software Engineering, Volume 27 Issue 10. DOI = 10.1109/32.962562
3) Praveen Ranjan Srivastava , 2005-2008. Test case Prioritization, of theoretical and applied information technology, Journal of theoretical and applied information technology
4) Bo Jiang, Zhenyu Zhang, T. Y. Chen, 2009. How Well Do Test Case Prioritization Techniques Support Statistical Fault Localization. Computer Software and Applications Conference, COMPSAC '09. 33rd Annual IEEE International, DOI = 10.1109/COMPSAC.2009.23
5) InfoDesign OSD GmbH. iValidator. Retrieved on 06 August 2010 from http://www.ivalidator.org
6) Siripong Roongruangsuwan, Jirapun Daengdej, 2005-2010. Test Case Prioritization Techniques. Journal of theoretical and applied information technology
7) Jefferson Offutt, Jie Pan, Jeffrey M. Voas, 1995. Procedures for Reducing the Size of Coverage-based Test Sets. Twelfth International Conference on Testing Computer Software, 111—123. Gorge Mason university computer science department.
8) David Hovemeyer, William Pugh, December 2004. Finding bugs is easy. ACM SIGPLAN Notice, Volume 39 Issue 12. DOI=10.1145/1052883.1052895
9) Colin Bird, Andrew Sermon, March 2001. An XML-based approach to automated software testing. ACM SIGSOFT Software Engineering Notes, Volume 26 Issue 2. DOI= 10.1145/505776.505792
10) Louridas, P, July 2006. Static Code Analysis. Software IEEE, Volume 23 Issue 4. 58-61. DOI= 10.1109/MS.2006.114
11) Hongyu Zhang; Xiuzhen Zhang; Ming Gu; Dec 2007. Predicting Defective Software Components from Code Complexity Measures. IEEE Dependable

Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on, 93 – 96. DOI=10.1109/PRDC.2007.28

12) Srivastava, Thiagarajan, July 2002. Effectively Prioritizing Tests in Development Environment. ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis, 22-24. DOI: 10.1145/566171.566187
13) Checkstyle, Retrieved on October 2010 from http://checkstyle.sourceforge.net/
14) Infoether. PMD. Retrieved on October 2010 from http://pmd.sourceforge.net/
15) Department of computer science, university of Maryland.Findbugs. Retrieved on October 2010 from http://findbugs.sourceforge.net/
16) Mark Doliner, 2005. Cobertura. Retrieved on October 2010 from http://cobertura.sourceforge.net/
17) Yu-Chi Huang, Chin-Yu Huang, Jun-Ru Chang and Tsan-Yuan Chen, 2010. Design and Analysis of Cost-Cognizant Test Case Prioritization Using Genetic Algorithm with Test History. Computer Software and Applications Conference (COMPSAC), IEEE 34th Annual, 413-418. DO=10.1109/COMPSAC.2010.66
18) Soyatec, 2006. eUML2. Retrieved on October 2010 from http://www.soyatec.com/euml2
19) Perforce,1996. Retrieved on October 2010 from http://www.perforce.com/
20) Gregory Tassey, 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. U.S. Department of Commerce National Institute of Standards and Technology.
21) Glenford Myers, 2004. The art of software testing. Published by John Wiley & Sons, Inc. ISBN: 0-471-46912-2
22) Hyunsook Do, Gregg Rothermel, Alex Kinneer, 2005. Prioritizing JUnit Test Cases:An Empirical Assessment and Cost-Benefits Analysis. Citeseerx Penn state university.
23) Gregg Rothermel, Roland H. Untch, Chengyun Chu, Mary Jean Harrold, 1999. Test Case Prioritization: An Empirical Study. Citeseerx Penn state university, In proceeding of the international conference on software maintenance.

24) Kostas Kevrekidis Stijn Albers, Peter J. M. Sonnemans, Guillaume M. Stollman, 2009. Software Complexity and Testing Effectiveness: An Empirical Study.IEEE Reliability and Maintainability Symposium. DOI=10.1109/RAMS.2009.4914733
25) Christian Hargraves, 2003. Jamelon. Retrieved on October 2010 from http://jameleon.sourceforge.net/index.html
26) Timothy Wall. 2008. Abbot. Retrieved on October 2010 from http://abbot.sourceforge.net/doc/overview.shtml
27) Alian Abram, James W Moore, Pierre Borque, Robert Dupuis, 2003. Software Engineering body of knowledge. ISBN: 0-7695-2330-7.
28) Atlassian Jira.JavaNCSS. Retrieved on October 2010 from http://jira.codehaus.org/browse/JAVANCSS

Appendix A: Glossary

Annotation in Java	:	Annotation in Java is a syntactic metadata that can be added to Java source code
Component	:	Component refers to a module that consists of related functions or data
Defect	:	Defect is a generalized representation of all of the following [27]: • Error: A difference between a computed result and the correct result • Fault: An incorrect step, process, or data definition in a computer program • Failure: The incorrect result of a fault • Mistake: A human action that produces an incorrect result
Defect severity	:	Defect Severity is a scale to indicate the degree of impact that a defect has on the system.
Fault	:	Fault is an incorrect step, process, or data definition in a computer program which causes the program to perform in an unintended or unanticipated manner [27].
Generics in Java	:	Generics in Java: is a built in language feature that will make software more reliable by allowing the operation of a method on objects of different types.
System under Test (SuT)	:	SuT refers to a software or system that is being verified in the testing cycle.
Test case	:	Test case is a document indicates the steps to be executed, inputs and expected output, to check compliance to a specific requirement.
Test Description	:	Test Description describes a scenario which is a sequence of activities in a use case or combination of sequences across multiple use cases
Test Environment	:	Test environment is hardware and software environment in which test cases will be executed [27].
Test Scenario	:	Test scenario is a definition of a set of test cases or test scripts and the sequence in which they are to be executed.
Test Script	:	Test script is commonly refers to the instructions to be carried out for a particular test by an automated test tool.
Test Step	:	Test Step describes a small sequence of activity in a use case

Test Suite	:	Test suite consists of multiple tests to verify and validate a behavior or requirements of a product.
Testing	:	Testing is a process of analyzing the features of the software item to detect differences between existing and required conditions.

Appendix B: XML schema for test description

The XML schema for test description in iValidator is as illustrated below [5]:

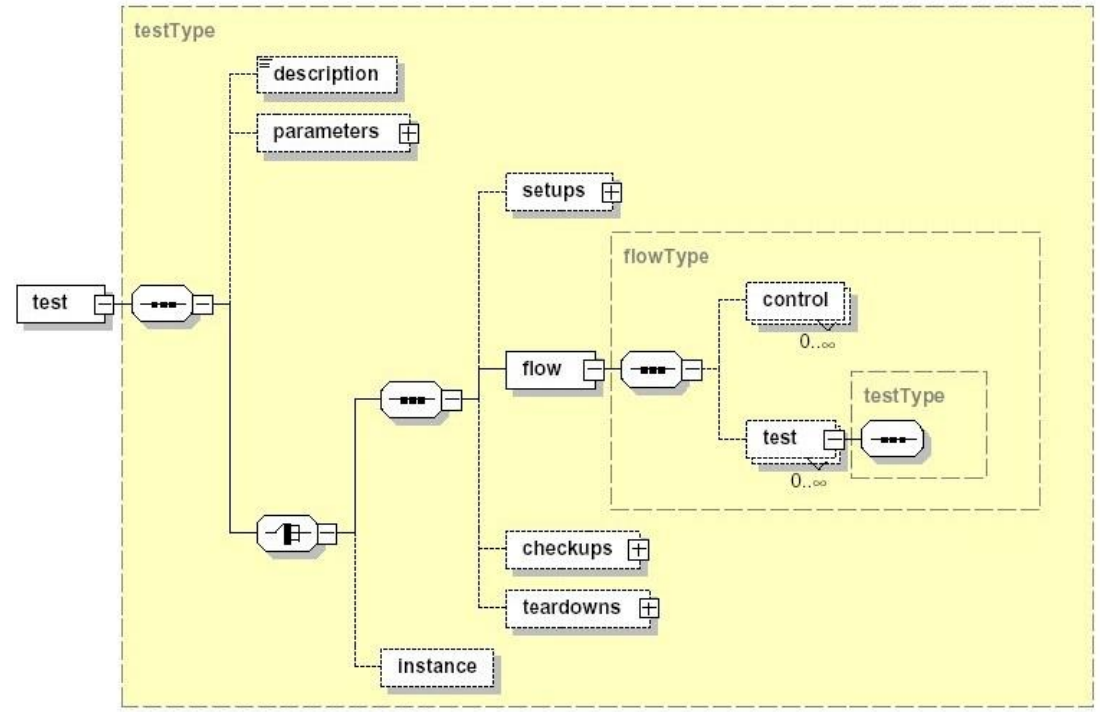

Figure Appendix B: Test description xml schema

The details of the schema are as described below:

Element	Description
Parameters	The parameters are test data to be passed to the test step or test description
Setups	Setup allows defining any initialization to be performed on system under test before test execution begins. It can refer to a test step or test unit.
Tear down	It represents the destruction of variables, adapters and parameters at the end of test execution. It ensures the test environment returns to the defined state.
Checkup	It indicates a condition that needs to be verified after test execution is completed.
Instance	Instance of a test step
Flow	Flow represents test steps to be executes in the test description. Each flow contains a single test step

Appendix C: XML schema for descriptor repository

The XML schema for descriptor repository in iValidator is as illustrated below [5]:

```
<descriptor repository="xml">
   <property name="inputpath" value="input"/>
   <property name="input" value="simple.xml"/>
   <property name="classpath" value="classes"/>
</descriptor>
```

The schema is described below:

Element	Description
Inputpath	It is the relative or absolute path for xml test descriptions
Input	Test description file name
Classpath	Path for test step classes

Appendix D: iValidator sample XML report repository

The sample XML report repository in iValidator as described below:

```
<report xmlns="http://www.ivalidator.org/schemas/xml-repository">
  <type>test</type>
  <name>TestLogin</name>
  <result>success</result>
  <start-time>2005-01-26T16:46:21.015+01:00</start-time>
  <end-time>2005-01-26T16:46:21.676+01:00</end-time>
  <duration>00:00:00.661</duration>
  <test>
    <type>test</type>
    <name>Successful_Login</name>
    <result>success</result>
    <start-time>2005-01-26T16:46:21.246+01:00</start-time>
    <end-time>2005-01-26T16:46:21.476+01:00</end-time>
    <duration>00:00:00.230</duration>
    <test>
      <type>test</type>
      <name>testLoginSuccessful</name>
      <result>success</result>
      <note></note>
      <start-time>2005-01-26T16:46:21.246+01:00</start-time>
      <end-time>2005-01-26T16:46:21.246+01:00</end-time>
      <duration>00:00:00.000</duration>
    </test>
  </test>
  </report>
```

The test description TestLogin consists of test steps testLoginSuccessful and Successful_Login. The report indicates the test description and test step start time, end time and duration of execution. <result> indicates whether the test step has passed or failed. The test description result is set to pass only if all the test steps pass or else it is set to fail.

Appendix E: XML schema for iValidator configuration file

The xml schema for configuration file is mentioned below:

```
<?xml version="1.0" encoding="UTF-8"?>

<config xmlns="http://www.ivalidator.org/schemas/ivalidator"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <descriptor repository="xml">
                <property name="input" value="test.xml"/>
                <property name="classpath" value="./;./bin/;./classes/"/>
   </descriptor>

   <report repository="xml">
                <property name="outputdir" value="./"/>
   </report>

</config>
```