Linnæus University
School of Computer Science, Physics and Mathematics

Degree Project

Capturing JUnit Behavior into Static Programs

Asher Siddiqui
2010-05-11
Subject: Computer Science
Level: Master
Course code: DA4004

Abstract

In this research paper, it evaluates the benefits achievable from static testing framework by analyzing and transforming the *JUnit3.8* source code and static execution of transformed code. Static structure enables us to analyze the code statically during creation and execution of test cases. The concept of research is by now well established in static analysis and testing development. The research approach is also increasingly affecting the static testing process and such research oriented work has proved particularly valuable for those of us who want to understand the reflective behavior of *JUnit3.8 Framework*.

JUnit3.8 Framework uses *Java Reflection API* to invoke core functionality (test cases creation and execution) dynamically. However, *Java Reflection API* allows developers to access and modify structure and behavior of a program. Reflection provides flexible solution for creating test cases and controlling the execution of test cases. Java reflection helps to encapsulate test cases in a single object representing the test suite. It also helps to associate each test method with a test object. Where reflection is a powerful tool to perform potential operations, on the other hand, it limits static analysis. Static analysis tools often cannot work effectively with reflection.

In order to avoid the reflection, *Static Testing Framework* provides a static platform to analyze the *JUnit3.8* source code and transform it into non-reflective version that emulates the dynamic behavior of *JUnit3.8*. The transformed source code has possible leverage to replace reflection with static code and does same things in an execution environment of *Static Testing Framework* that reflection does in *JUnit3.8*. More besides, the transformed code also enables execution environment of *Static Testing Framework* to run test methods statically. In order to measure the degree of efficiency, the implemented tool is evaluated. The evaluation of *Static Testing Framework* draws results for different Java projects and these statistical data is compared with *JUnit3.8* results to measure the effectiveness of *Static Testing Framework*. As a result of evaluation, *STF* can be used for static creation and execution of test cases up to *JUnit3.8* where test cases are not creating within a test class and where real definition of constructors is not required. These problems can be dealt as future work by introducing a middle layer to execute test fixtures for each test method and by generating test classes as per real definition of constructors.

Keywords

JUnit3.8 Framework, Test Case, Test Object, Test Method, Reflection, Test Fixture, Test Suite, Test Pack, Dynamic Behavior, Static Behavior, Static Analysis, Code Generation Environment, Execution Environment, Test Object Hierarchy, Assertion, Result Parameters, Source Code Transformation, Building Process, Test Case Execution, Potential Operations, Java Reflection API, Java Object Model, Annotation, Meta-Model, Call Graph, Reflection Resolution, BDD Database

Acknowledgments

I am thankful to almighty God who gave me strength to complete this project. Secondly, I completed thesis project accompanying prayers of my parents who always motivated and supported me for my entire study period.
I would like to pay gratitude to PhD student Tobias Gutzmann for providing me supervision through out the thesis during practical implementation of STF and in writing a good thesis report.

Content

Chapter 1
Introduction

The *JUnit3.8 Framework* uses dynamic structure for creating and executing the test cases. Each class containing test methods and instance of such class holding a test method is considered to be a test case. *JUnit3.8 Framework* creates instances, invokes constructors and test methods dynamically by using reflection. Reflection provides flexible solution for creating test cases and controlling the execution of test cases. Java reflection helps to encapsulate test cases in a single object representing the test suite. It also helps to associate each test method with a test object. Where reflection is a powerful tool to perform potential operations, on the other hand, it limits static analysis. Static analysis tools often cannot work effectively with reflection.

In order to overcome this obstacle, *Static Testing Framework* provides static platform to create the test cases and to control the execution of test cases. There are two domains working in *Static Testing Framework.* One is code generation environment and other is execution environment. The execution environment keeps record of results for each test case during the execution and it also displays drawn result. It addresses a possible set of goals important to analyze and transform the dynamic behavior of *JUnit3.8 Framework* and execute transformed code statically. *Static Testing Framework (STF)* draws results which display test methods and their corresponding results. It also keeps track of errors, when a value or reference in the test method refers to null then it is said to be some error in the test method.

1.1 Problem Definition

In *JUnit3.8 Framework*, testing technique is deployed dynamically using *Java Reflection API*. The reflection works for *JUnit3.8*, when test objects are created and test methods associated with test objects are executed [2]. The process of creating test objects and their association with test methods refer to building process of test cases. The problem with *JUnit3.8* reflective behavior is that building process and execution of test cases can not be analyzed statically to see how they work and to understand the collaboration of test cases during execution.

Reflection performs many potential operations. When a test case is built, the process begins from a test class and ends in form of a test case holding single test method of test class. When a test case is executed, test method referred by test case is invoked dynamically using reflection. Reflection makes *JUnit3.8 Framework* automate to create test cases and to control the execution of test cases. Java reflection provides facility to build test suite which can contain a single or multiple test cases. Java Reflection is very powerful concept to access and modify the structure and behavior. Most often, it is not used where static analysis is required for optimization and error detection. Reflection does not allow static access to its potential targets. Whereby, reflection performs potential operations that are out of access and cannot be analyzed. In order to analyze the testing process statically, we need to develop a static platform to build and execute

test cases statically by emulating the reflective behavior of *JUnit3.8 Framework*. It can make possible to invoke test methods associated with a test cases without using reflection.

1.2 Goal Criteria

Our goal criteria focus on the following perspective targets:

a) We need to find a way to emulate the *JUnit3.8* dynamic behavior into non reflective behavior.
b) We need to find a tool that can access the *JUnit3.8* source code which will be required to transform *JUnit3.8* source code.

1.3 Restriction to JUnit3.8

JUnit3.8 is an extensible framework to write single or multiple test cases. The test cases refer to explicitly test specific area in the Java program. The implemented approach for static analysis is restricted to *JUnit3.8*, it has possible capability to capture *JUnit3.8* dynamic behavior. The *JUnit3.8* is still widely used for unit testing and it is employed to test a hierarchy of Java programs. The *JUnit3.8* is still popular for unit testing, each program or module can be recognized easily as part of the test domain.

1.4 Thesis Outline

Thesis outline gives an overview of this report by describing different chapters briefly. *Chapter 1*, it describes reflection and static analysis problems due to reflection in the existing system. It also covers necessary requirements that motivate to implement a new static system and define goals need to be achieved. *Chapter 2*, it is necessary to understand *JUnit3.8 Framework* with intention to capture reflective behavior. It describes *JUnit3.8 Framework* code segments in detail that use reflection. It focuses design and implementation approach of *JUnit3.8 Framework* for building and execution of test cases. It also takes new concepts of writing test cases come with *JUnit4.0*. This chapter gives an overview of *Recoder0.94c* library. It describes model of *Recoder0.94c*, overview of its functionality and meta-programming. *Chapter 3*, the implemented approach is discusses in this chapter. It consists of detailed design and implementation of *Static Testing Framework*. It describes a model of *STF* in which each artifact is discussed in detail. It describes how *STF* is providing alternatives for *JUnit3.8 Framework* reflective features and aspects of *STF* methodology for code generation. It compares *STF* static behavior with *JUnit3.8 Framework* dynamic behavior. *Chapter 4*, it is necessary to evaluate the effectiveness of implemented approach to find the degree of achievement. Different projects and results drawn from testing procedures are discussed in this chapter. *Chapter 5*, it describes summary and conclusion of *STF*. It also describe what more can be expected from *STF* in future. *Chapter 6*, this chapter describes an approach to find solution to capture dynamic behavior of *JUnit3.8 Framework*. It describes some practical aspects of given approach.

Chapter 2
Background

This chapter briefly describes design techniques, functionality and working principles of *JUnit3.8 Framework*. It is very important to understand the working of *JUnit3.8 Framework*. It reveals problems due to which *Static Testing Framework* is deployed. *Java Reflection API* [15] provides a platform through which Java can expose the features of a class at runtime. A class, methods, fields and constructors can be accessed as objects and these objects reflect the *Java Object Model*. These objects can be used to access the features of corresponding class at runtime. The following examples of different reflective operations describe potential functionality that limits static analysis. An instance of a given class can be created at run time as shown in listing 2.1.

```
Class thisClass = Class.forName(“classnane”);
object = thisClass.newInstance();
```

Listing 2.1: Instance creation of a named class by suing reflection

Method can be invoked dynamically using reflection as shown in listing 2.2. We just need to give the name of the method and its parameters to invoke it.

```
Class myClass = Class.forName("classpath");
Class[] parameter = new Class[]{};
Method method = myClass.getMethod("showResults",
parameter);
Object[] argList = new Object[]{};
Object object = myClass.newInstance();
method.invoke(argList);
```

Listing 2.2: Method Invocation of named class by using reflection

Reflection also gives control on private members of a class. A user can access the fields of a class as mentioned in listing 2.3.

```
Secret instance = new Secret();
Class secretClass = instance.getClass();
Field fields[] = secretClass.getDeclaredFields();
```

Listing 2.3: Accessing set of fields in multiple methods presented in a class

2.1 JUnit3.8 Framework

JUnit3.8 Framework is a unit testing framework for the programming applications developed on *Java Platform* [1]. *JUnit3.8 Framework* is being developed by *Kent Beck and Eric Gamma* since 1997 and inspired by *SUnit (SmallTalk Testing Framework)*. It is useful to find the bugs in a Java application, not only find the bugs instead it also gives complete trace of bugs. Each finding bug tells that what are the expected results and what are actual results.

2.1.1 Features

JUnit3.8 Framework provides a versatile platform for unit testing of different Java applications. It has following features [10] which describe its importance for unit testing.

a) *JUnit3.8 Framework* saves time for developers to eliminate duplicate efforts. *JUnit3.8 Framework* provides platform for both development and testing.
b) *JUnit3.8 Framework* tests can retain their value over long time.
c) A person other than developer can also write test cases easily.
d) *JUnit3.8 Framework* gives facility to create new tests on existing tests.
e) Developers have test cases in their mind and they realize them in different ways like print statements, debugging expressions and test scripts. To avoid this type of testing, *JUnit3.8 Framework* provides one standard way to write tests.

2.1.2 Design Technique of JUnit3.8 Framework

JUnit3.8 Framework has collaboration based design [11] because objects play multiple roles in different context. Each test method is represented by an instance of a test class by which it is associated and each test class must be subtype of *TestCase* or *TestSuite* [3]. *JUnit3.8 Framework* follows two different design patterns simultaneously.

a) Command Pattern

An instance of class *TestCase* is a command object. A Command Pattern [11] creates one instance of a test class for an operation and assigns it that operation where instance of class *TestCase* encapsulates a single instance of test class [4]. The listing 2.4 shows the command Pattern.

```
static TestCase test1 = new TestClass("Test Method") {
        public void runTest() {
                testXXX();
        }

}
```

Listing 2.4: A test case representation as Command Pattern [8]

b) Composite Pattern

An instance of class *TestSuite* is a composite object [11]. It creates several instances of one or more test classes for several operations and assigns them those operations where instance of class *TestSuite* is representing one or more instances of class *TestCase* encapsulating one or more instances of test classes [4] (test cases) as shown in listing 2.5.

```
public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(test1);
        suite.addTest(test2);
        suite.addTest(test3);
        return suite;
}
```

Listing 2.5: A test suite representation as Composite Pattern [8]

2.1.3 JUnit3.8 Collaboration Based Design

JUnit3.8 Framework is designed on *Configurable Client-Server Collaboration Design*. A complete test run has two step process is as under:

a) Configuration

During configuration, a hierarchy of test objects is built in which one object holding a test method, each such object represents one test case. Basically, these objects are instances of subclass(es) of class *TestCase*, each method that starts with the name of "test" in the subclass(es) leads to an object in the hierarchy and such type of objects are leaf nodes in the hierarchy. Intermediate and root nodes in the hierarchy are instances of class *TestSuite* which provide group functionality [11].

b) Test-Case Execution

After building the test object hierarchy, the execution of actual test case is started by calling run on the root object of the test object hierarchy. The design implementation of *JUnit3.8 Framework* describes hierarchical approach to execute the actual test method. The approach is said to be hierarchical because class *TestSuite* is subtype of interface *Test*, an instance of class *TestSuite* encapsulates one or more instances of class *TestCase* and each single instance of class *TestCase* encapsulates single instance of test class (test case) [5]. So this type of object creations provides tree like structure. The purpose of building this hierarchy is to wrapping each test case in monitoring instances. The root object refers to the instance of class *TestSuite* that is built during the configuration process. Traversing technique on the object hierarchy is *Depth-First*. The order of the each object or node in the test object hierarchy is determined by the order in which they were added to the tree. During execution, a run is called on every leaf of the test object hierarchy [6] that is instances of subclass(es) of class *TestCase* [7] as shown in Figure 2.1. The result of each test case is recorded by an instance of class *TestResult* which collects resulting parameters. The instance of class *TestResult* is passed on from one test case to another in the test object hierarchy to collect the comprehensive detail of executed tests.

Each test case is also wrapped in test fixtures. The test fixture *setUp()* is called before execution of each test case and test fixture *tearDown()* is called after execution of each test case. It means actual test case is wrapped by calling the methods *setUp()* and *tearDown()*. These methods can be overridden to describe the initialization and de-initialization of test case. If they are not over ridden in test class, default *setUp()* and *tearDown()* are called. The default definition of test fixtures has empty body [11].

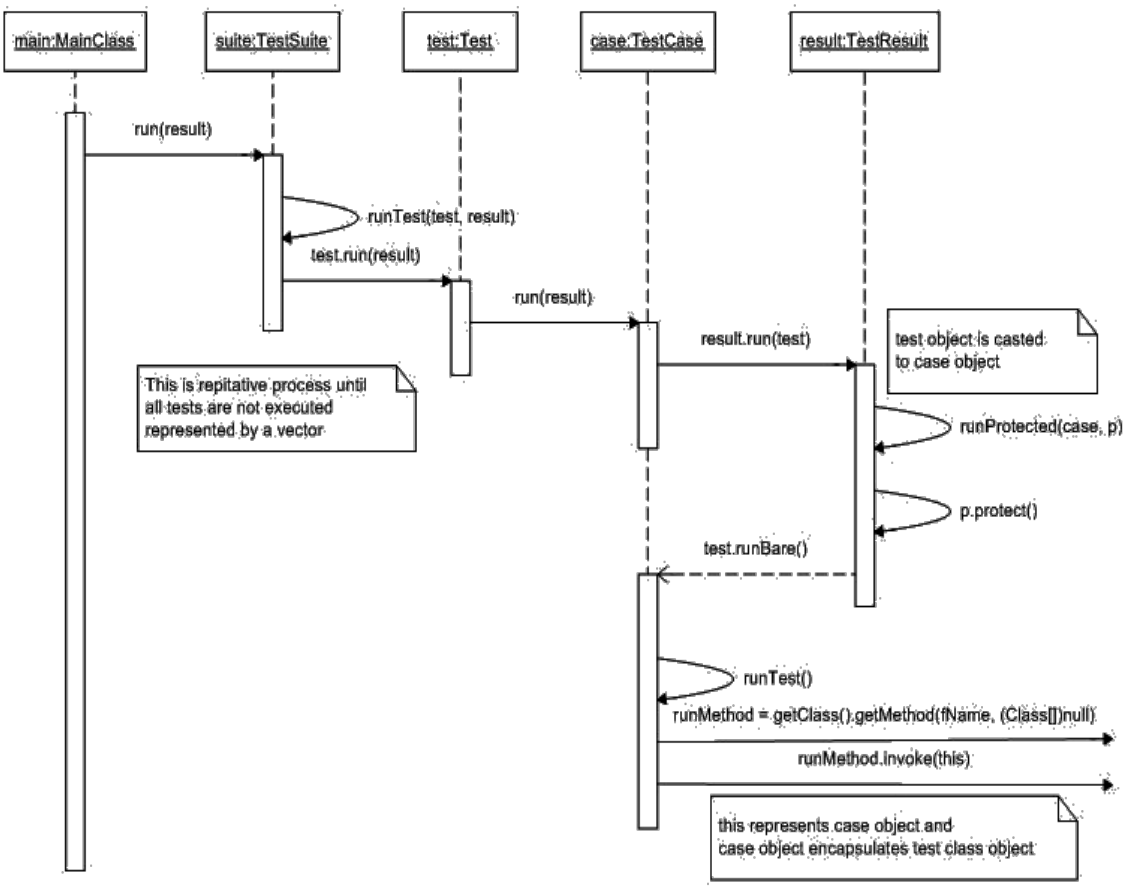

Figure 2.1: Sequence Diagram of Test Case Execution

2.1.4 Errors and Failures Handling

An instance of class *AssertionFailedError* is created when a test method is failed. The errors occur when something is going wrong. In *JUnit3.8 Framework*, errors and failures are handled as exceptions. These exceptions are collected by instance of class *TestResult*. When a test object is failure or there is some error in the test method, it wraps into an instance of class *TestFailure* with complete stack trace. Each test object is stored in a list of failure tests if it is declared as failure or in an error list if there is some error in the test method [11].

In test hierarchy, when a *run()* method of class *TestCase* is called on leaf, it does not call test method directly. First this job delegates to the instance of class *TestResult* as an argument and class *TestResult* prepares some pre-requisites and then call back on test case for the test fixtures and actual code execution [11].

The pre-requisites are

a) Protectable Interface defines a test failure protocol.
b) It notifies about the start and end of test case execution.

2.1.5 JUnit4.0

JUnit4.0 aims to simplify *JUnit3.8 Framework*, the purpose of *JUnit4.0* is to enable developers to write more test cases. It has still backward compatibility with *JUnit3.8*. Test methods and test fixtures are identified by naming conventions up to *JUnit3.8*. *JUnit4.0* uses annotations to identify test methods and test fixtures. The evolution of different versions of *JUnit Framework* introduced some extra features and capabilities but *JUnit4.0* provides more simple and incremental platform as compared to previous versions. In *JUnit4.0*, each class is considered to be a test class that contains test methods, there is no longer need to extend each test class from class *TestCase*. Each test class can be constructed independently, hereby it is easy to write test cases in *JUnit4.0*.

A method is considered to be a test method if it has *@Test* annotation as shown in listing 2.6, the name of test method does not need to start with test, e.g.,

```
@Test public void addTest() {
        int z = x + y;
        assertEquals(2, z);
}
```

Listing 2.6: A test method definition with test annotation

Each method with annotation *@Before* is considered as test fixture that must be executed before execution of actual test methods and each method with annotation *@After* is considered to be test fixture that must be executed after execution of actual test method as shown in listing 2.7. There is no restriction to define number of test fixtures, e.g.,

```
@Before protected void initializeDirectory() {
        dir = new File("path");
        dir = new File(dir, "xslt");
        dir = new File(dir, "input");
}
@After protected void redirectStderr() {
        dir.close();
}
```

Listing 2.7: A test fixtures definition with test fixture annotation

2.2 Recoder0.94c

Recoder0.94c is a tool that is developed on Java platform. It has property to provide meta-model of a Java project as shown in Figure 2.2. The design of *Recoder0.94c* describes a parser that can parse Java project and an analyzer to analyze the parsed data. It also provides facility to convert meta-model back into source code [12].

Once *Recoder0.94c* builds meta-model, it provides a service to access and modify source code comes from meta-model. It also provides services to transform source code as per requirements. The design of *Recoder0.94c* [12] makes possible to analyze source code of given Java project. A meta-model contains complete information of entities presented in a system, their elements and the relationship between them.

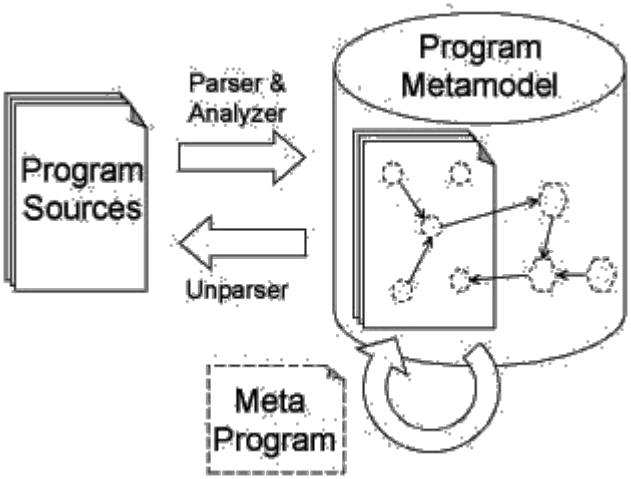

Figure 2.2: Static Meta Programming of Java Source [12]

2.2.1 Meta-Programming

In general, meta-program is a program that can access and manipulate other computer programs. It uses meta-model information for transformations of source code. The meta-programming [14] provides access to the source code and the transformation of source code makes possible to emulate the reflective behavior into static behavior. It can transform programs into more flexible and efficient source code required by the system to handle different prospective.

2.2.2 Transformation

Recoder0.94c provides set of transformation tools. In order to deploy transformation services, we need to set the input path for project settings of cross reference service configuration. It takes all the Java files found in input path and transform them. The Transformation can perform set of operations on the source code according to the requirements. The transformation services provide control over each type [12], their elements and their relationship.

2.2.3 Recoder0.94c Program Model

Recoder0.94c provides meta-model of given project. The information from the meta-model can be accessed and modified. *Recoder0.94c* provides some special services that come in action if a user makes changes into the source code, the corresponding meta-model is updated automatically in the database. As Figure 2.3 shows, a meta-model resides in a database, the most

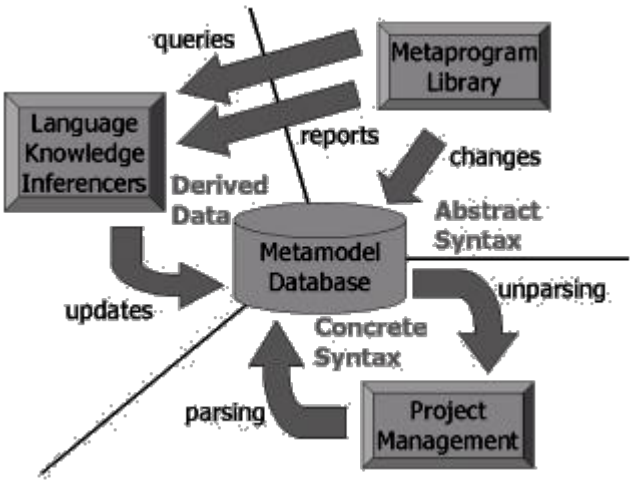

Figure 2.3: *Recoder0.94c* Program Model [12]

interactive part is *Project Management*. It triggers the building process of meta-model for given project. *Meta-Program Library* is used to access and modify meta-information from the meta-model and then it generates reports about changes which have been made. The changes must follow rules and regulations of particular language *(Syntax & Semantics)*. The changes consist of expressions and predicates that must be computed by *Language Knowledge Inference*. The *Language Knowledge Inference* ensures that changes are made according to the syntax and semantics of language, it updates meta-model in the database.

2.2.4 Features

The main features [13] of *Recoder0.94c* give different application aspects:

a) Parsing and un-parsing of Java sources

Recoder0.94c derives a model representing entities found in the source code or class files. This meta-model refers to a highly detailed syntactic model without any loss of information. *Recoder0.94c* gives facility to retain commenting and formatting by using corresponding transformation service. *Recoder0.94c* does comprehensive parsing of Java source code. *Recoder0.94c* pretty printer service gives source code of a program like classes and their elements. Source code can be reproduces and new code can be embedded as well. This feature is useful for performing simple processing and reproducing source code as per requirements.

b) Name and type analysis for Java programs

Recoder0.94c provides a quick Java source code frontend including name and type analysis as well as cross reference information. Transformation services give this facility by resolving references. It can evaluate compile time constants. Software based on Java can be analyzed in terms of type analysis. Name and type analysis feature provides number of semantic checks.

c) Transformation of Java sources

It can transform *Abstract Syntax Tree (AST)* into source code easily, including undo functionality and partially incremental model update. *Recoder0.94c* extracts information from *Abstract Syntax Tree (AST)*, it has ability to transform *AST* into simple information.

d) Incremental analysis and transformation of Java sources

Recoder0.94c itself provides set of services to build and update model automatically. The model updating can be done incrementally. The benefit of incremental updating is to extend the meta-programming library. Transformation services take care of dependencies when it performs update to local data. *Recoder0.94c* analyzes change in its model and performs update automatically.

Chapter 3
Approach/Implementation

As mentioned in the chapter 2, *JUnit3.8 Framework* creates instances of test classes dynamically, associates them with test methods containing actual code, stores them in a list and then follows hierarchical approach during dynamic invocation of test methods. One method calls other in the hierarchy. Finally actual test method is executed and this process repeats until all test objects are executed.

This chapter describes structure and design theory of *Static Testing Framework* and also explains how does *STF* resolves reflective issues for static analysis as mentioned in *JUnit3.8 Framework*. *Static Testing Framework* implements different approach which converts dynamic behavior of *JUnit3.8 Framework* into non-reflective behavior. *STF* introduces two domains such as code generation environment and execution environment as shown in figure 3.1.

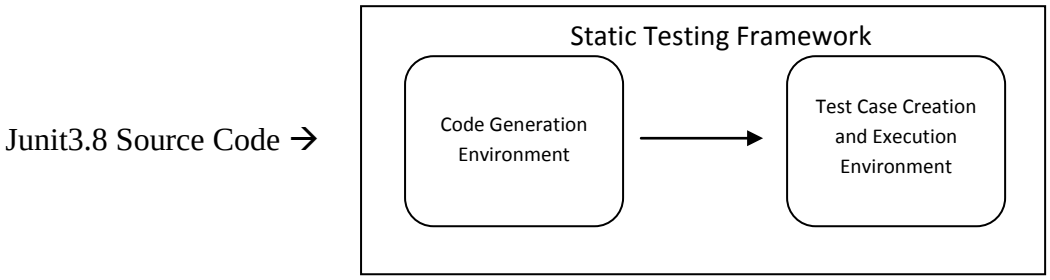

Figure 3.1: Block Diagram of STF

As the name indicates, code generation environment generates test classes by transforming *JUnit3.8* source code into static version. The code is transformed according to the execution environment of *STF* and execution environment is responsible to create and execute test cases. *STF* deals with errors in a different way, some of the errors are *Code Generation Time Errors* that are encountered during code generation like a test class does not contain valid constructor or public modifier and a test method is not public. Some of the errors are *Run Time Errors* that are encountered during execution of test methods like if any reference or value refers to null. The code generator displays simply a message when a *Code Generation Time Error* occurs but *Run Time Errors* are referred to errors occur during execution of test cases and treated as exceptions.

3.1 Design Structure of Static Testing Framework

The design describes *Static Testing Framework* in two parts with respect to code generation environment and execution environment.

3.1.1 Static Diagram of Code Generation Environment

In order to analyze *JUnit3.8* source code, there is need of such a tool that provides meta-model for *JUnit3.8* source code. The meta-model enables us to analyze and manipulate *JUnit3.8* source code according to requirements. The diagram Figure 3.1 consists of a

single class named *CodeGenerator*. It is very important part of *STF*. The execution environment absolutely depends on code generator. The code generator uses *Recoder0.94c Library* to generate non-reflective version of test classes. *Recoder0.94c* provides access over source code of one or multiple Java files. It contains meta-models which is representation of Java programs. The purpose of *JUnit3.8* source code analysis and transformation is to make it acceptable for static execution of test cases.

a) CodeGenerator

As the name indicates, it generates non-reflective code by transforming *JUnit3.8* source code. The access and analysis of *JUnit3.8* source code is possible by using *Recoder0.94c Library*. It is providing wide range of transformation services. It generates test classes again in a smart and meaningful way by modifying and adding some extra functionality.

The class *CodeGenerator* sets up following paths:

- *Recoder0.94c* library, *JUnit3.8 Library* and source of Java project to set project properties of class *CrossReferenceServiceConfiguration*.
- Directory in which tester needs to write the generated code.
- *JUnit3.8* source code of Java project consists of test classes.

This service allows code generator to have an access over source code for all test classes inherited from class *TestCase* and class *TestSuite*. The class *CodeGenerator* comes with set of checks for *Code Generation Time Errors* as described in introduction of chapter 3.

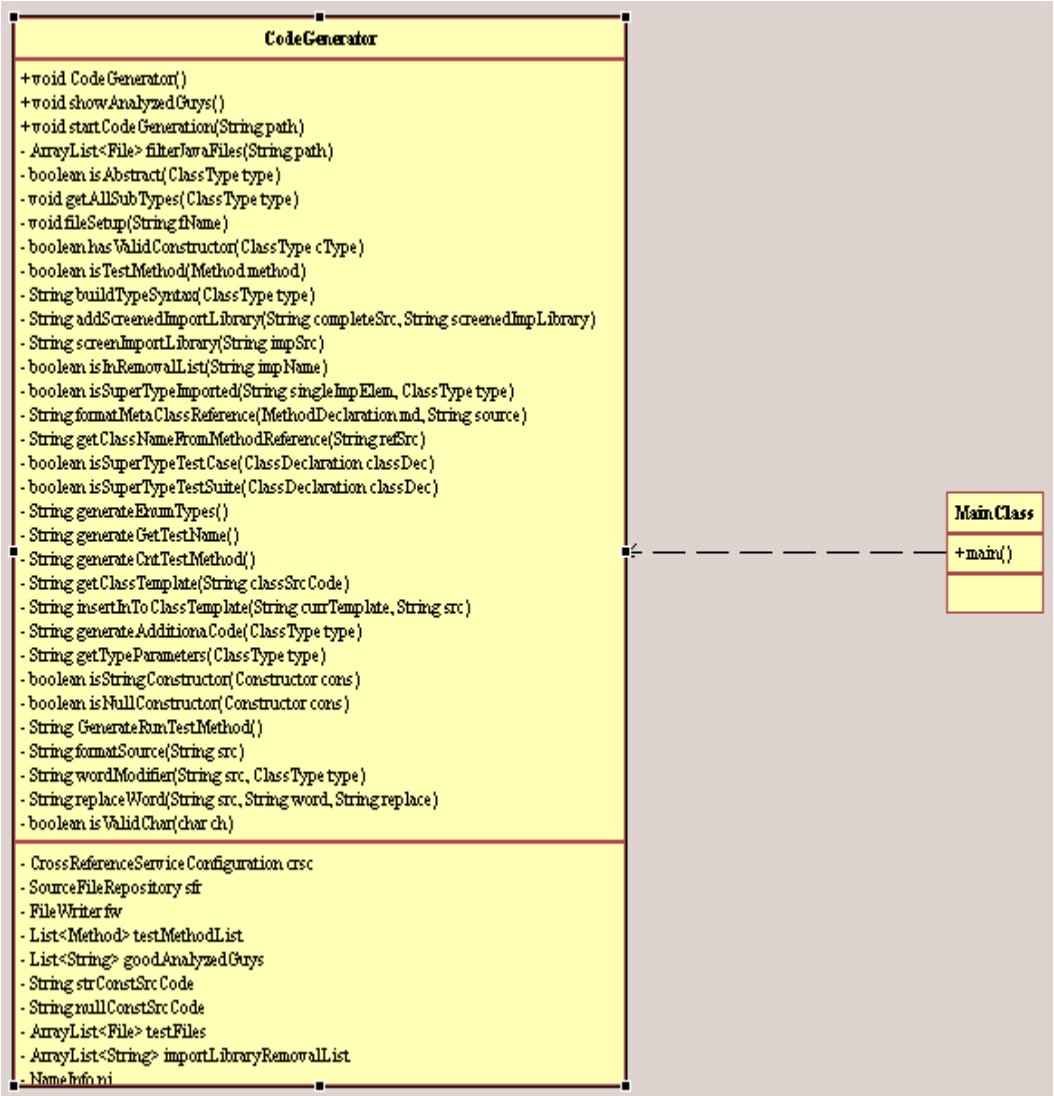

Figure 3.2: Static Diagram of Code Generation

If a test method does not fulfill criteria (its return type must be void, must not have any parameter and its name must start with word “test”), it will not be included in the test class needs to be transformed. In order to get nice syntax of generated classes, it is having lot of formatting methods to transform the source code in acceptable format. The source code of each test method needs to be formatted. *Recoder0.94c* library gives access over source code of every atomic element presented in the type. The code generator keeps track of test methods and endorses this information in each test class by adding a constant data member *testMethodCounter*. It generates new test class for each qualified subtype of class *TestCase* and class *TestSuite*. In fact, code generator provides a generic pattern as shown in Figure 3.1 to generate test classes but this pattern has ability to change transformation automatically according to the class modifier (abstract or public). Some common member functions are generated for each test class, having same prototype but different definition. The test methods are generated with the same code as written in *JUnit3.8* source code.

3.1.2 Static Diagram of STF Execution Environment

The execution environment of STF provides a static structure to create instances of test classes and it also invokes test methods statically.

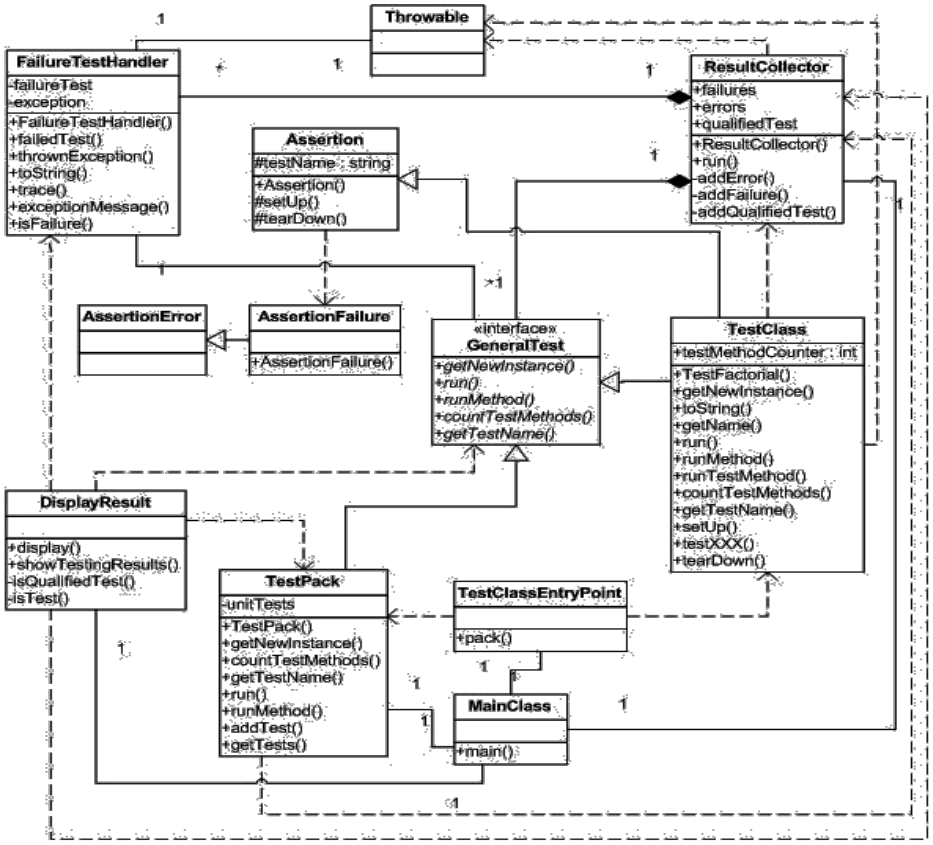

Figure 3.3: Static Diagram of Execution Environment [9]

The building process of a test case refers to creation of an instance of test class for a test method and assigns it that test method. The code is transformed according to the requirements of static structure to build and execute the test cases. It enables the transformed code to run independently without using *JUnit3.8* library by emulating the dynamic behavior. The execution environment provides complete report after building and execution of test cases. Each test failure or some error in a test method refers to complete stack trace. The collaboration of instances of different classes and their relationship is defined in Figure 3.2:

a) TestClass

One of the major problems in conversion of dynamic reflection into non-reflective behavior is to associate an instance of a test class with a particular method. The generated class is more functional, It is providing information about names of test methods and It is also providing member functions to associate instances with particular test methods. Each test class has property to return its instance statically. A test class contains a method *run(ResultCollector result)*, the purpose of this method is to encapsulate the test object into an instance of class *ResultCollector*. The actual test method execution begins when

test class method *runMethod()* is called from *run(GeneralTest test)* method of class *ResultCollector*. The method *runMethod()* calls *runTestMethod(GeneralTest test)* containing *setUp()* and *tearDown()*. It has ability to catch exceptions and throw them again. The method *runTestMethod(GeneralTest test)* selects the actual test method among number of test methods because it checks each test method to find a particular test method and these checks added at the time of code generation.

A test class contains enumerated data type to store names of qualified test methods existed in test class. The set of test method names is useful during building process of test pack. Each instance of test class has an attribute *testName* to set name of test method extracted from enumerated type and this name refers to the particular method name. The selection criteria of a test method works on the basis of test object attribute *testName*. So, the number of checks must be equal to the number of qualified test methods. Each test class inherits from class *Assertion*. The reason is that each test class is having test methods and test methods are evaluated on the basis of assertion methods. The assertion methods come from class *Assertion* to compare the expected and actual values. An *Assertion* class is having default test fixtures that are over ridden in test classes. Each test class implements interface *GeneralTest* to make handling of test objects easy as shown in Figure 3.2.

b) TestClassEntryPoint

The test pack building process begins from class *TestClassEntryPoint*. The method *pack()* allows us to pass instances of test classes need to be included in test pack as mentioned in Figure 3.2. Test pack building process refers to the creation of test class instances, associate them with particular methods and store them in a vector. Ultimately, *pack()* method returns an instance of class *TestPack*, containing test cases. The instance of class *TestPack* will be root of building hierarchy.

c) TestPack

It contains a data member vector to store test objects. The tests cases belong to the different test classes are stored in their corresponding vector lists. Each instance of class *TestPack* refers to its own vector containing all the test cases. Test method execution triggers when the *run(ResultCollector result)* method of class *TestPack* is called. A test pack is built when *addTestPack(GeneralTest test)* method is called. When an instance of test class is created, a test method is assigned to it by giving name of test method and it gets name of test method from the enumerated list of test method names contained by each test class. Finally these test cases are added into the vector. Test pack is built by calling the *pack()* method of class *TestClassEntryPoint*. Each test object in the class *TestPack* refers to the interface *GeneralTest*.

d) GeneralTest

It is an interface and provides easiness to handle test objects during building process and test cases execution. Each test case is treated as reference to interface *GeneralTest*. It is containing a set of abstract methods that each class needs to implement as shown in Figure 3.2. Each test class defines abstract methods and calls them during building process and execution of test methods.

e) ResultCollector

It is responsible to collect the result of all test methods during execution either it is passed or failure. It has lists to store passed and failure tests separately. Each failure test is handled as an instance of class *TestFailureHandler* and each qualified test is handled as an instance of test class. The method *run(GeneralTest test)* provides shell to encapsulate the test method, catches exception of particular type and adds it into that type of list. There are some add method to add test failures, qualified tests and tests having errors as mentioned in Figure 3.2.

f) TestFailureHandler

The instance of class *TestFailureHandler* keeps complete track of failure test method or test method in which an error occurs. It holds the name of failure test or test method in which an error occurs and the exception thrown during the execution of test method with its complete stack trace. This stack trace helps the user to find the failure test method or test method having errors as mentioned in Figure 3.2.

g) Assertion

The class *Assertion* is most important because it makes test methods result oriented. It decides whether a test method is failure or passed. It has several assertion methods to handle different data types as shown in Figure 3.2. It contains *setUp()* and *tearDown()* methods that can be overridden in test classes. For each assertion method, if a test method is failure, a method *fail(String msg)* is called and a new instance of class *AssertionFailure* is thrown as an exception. The instance of class *AssertionFailure* is finally caught by the *run(GeneralTest test)* method of class *ResultCollector*. The Assertion class is containing some private assert methods that take message and this message is formatted if a method is failure as mentioned in Figure 3.2. The method *buildMsg(String msg, Object expected, Object actual)* returns a message as a result to describe what actual value is and what expected value is?

h) AssertionFailure

The instance of class *AssertionFailure* is created when a test method is failure. The class *AssertionFailure* is inherited from class *AssertionError* as shown in Figure 3.2. The class *AssertionError* is a built-in Java class that extends from class *Error* and class *Error* extends from class *Throwable*. The purpose of this

chain of hierarchy is to throw an instance of class *AssertionFailure* as an exception when a test method is failure.

3.2 JUnit3.8 Features Handling

This section describes solutions for *JUnit3.8* dynamic features that needed to convert into static behavior for static analysis. The following prospects describe core dynamic features and their solution:

3.2.1 Initialization of Building Process

In *JUnit3.8*, class *TestSuite* has a method *addTestSuite(Class.class)* that takes test class as parameter and then it performs reflective operations on it to create instances, to invoke constructors and to invoke test methods.

In our case, there is a class *TestPack* which is responsible for building complete package of test cases. The class *TestPack* has a method *addTestPack(Object object)*, it takes object of test class as parameter rather than class itself. *STF* design does not allow passing test class as parameter. It does not have such a structure to get instances of passing class.

3.2.2 Creation of Test Object

The technique used in *Static Testing Framework* to create the instances of test classes is rather tricky. During the code generation process, code generator adds some additional member functions that help to create test objects statically. So, here is a method *getNewInstance(String name)* that returns instance of test class of which object is passed to the *addTestPack(Object object)* method. Therefore, each generated test class is inherited from class *Assertion* and implements interface *GeneralTest*, the instance of test class refers to interface *GeneralTest*.

3.2.3 Association of Single Test Method with Single Test Object

There is an enumerated list contains names of test methods existed in each test class. The enumerated variables are created for only those test methods that meet the criteria to being a test method.

```
enum Tests {
        test1, test2
};
```

Listing 3.1: Enumerated list of test methods containing test names

Each test class has a constant data member *testMethodCounter* that represents the number of test methods need to be executed. This variable is defined during code generation. Here is a method *getTestName(int position)* that takes position of the enumerated variable and then returns it as mentioned in listing 3.1.

Now we have test method names and number of test methods. In the *addTestPack(Object object)* method, a loop iterates up to value of *testMethodCounter*. The instances of test class are created under this loop, *getTestName(int position)* returns the test method name by passing loop variable as parameter.

3.2.4 Execution of Test Cases

After successful creation of test objects and their association with test methods, each test case is added to the vector list and ready to execute. The class *TestPack* contains a method *run(ResultCollector result)* that triggers the execution of test cases. The method *run(ResultCollector result)* contains loop that takes test cases from vector list one by one and executes them as shown in listing 3.2.

```
public void run(ResultCollector result) {
        for (GeneralTest test : unitTests) {
                test.run(result);
        }
}
```

Listing 3.2: Execution of test cases contained by the vector

Here a new class *ResultCollector* is emerging, it has also a method names *run(GeneralTest test)*. In the test class *run(ResultCollector result)* method, result object calls its own *run(GeneralTest test)* method as shown in listing 3.3. In *run(GeneralTest test)* method, test case calls its own *runMethod()*.

```
public void run(ResultCollector result) {
        result.run(this);
    }
```

Listing 3.3: Control passes to instance of class *ResultCollector*

The *setUp()* and *tearDown()* are fixtures of test methods, *setUp()* must be called before execution of test method and *tearDown()* after execution of test method as shown in listing 3.4. The method *runTestMethod(Object object)* calls the actual test method and execute it.

```
public void runMethod() throws Throwable {
        Throwable excep = null;
        setUp();
        try {
                runTestMethod(this);
        }
        catch(Throwable runTestExcep){
                excep = runTestExcep;
                throw excep;
        }
        finally {
                try {
                        tearDown();
                } catch (Throwable tearDownExcep) {
                        if (excep == null)
                        excep= tearDownExcep;
                }
        }
        if (excep != null)
        throw excep;
}
```

Listing 3.4: Test methods execution with test fixtures

3.2.5 Recognition of Own Test Method by Test Objects

A test object specifies an attribute *testName* to represent the name of test method. So test objects are associated with test methods on behalf of this attribute. There is a method *runTestMethod(GeneralTest test)*, code generator writes the appropriate number of checks according to the number of test methods. The method *runTestMethod(GeneralTest test)* compares test object attribute *testName* with the name of each test method called in predefined conditions as mentioned in listing 3.5. Each condition refers to the execution of particular test method. The match of *testName* with the name of test methods tells that there is an association between this test method and test case so, this test method is executed.

```
public void runTestMethod(GeneralTest test) throws Throwable {
        try {
                if (test.toString().equals("test1")) {
                        testDummy();
                }
                else if (test.toString().equals("test2")) {
                        testFact();
                }
        } catch (Throwable e) {
                e.fillInStackTrace();
                throw e;
        }
}
```

Listing 3.5: Selection of right test method according to its name

3.2.6 Collecting Result Parameters

STF adopts same technique to collect resulting parameters as *JUnit3.8* does. To avoid the execution of test methods directly, each test case is wrapped into an instance of class *ResultCollector* as shown in listing 3.6. The instance of *ResultCollector* passes on from one test method to another and keeps track of results for each occurrence of error and test failure.

```
public void run(GeneralTest test) {
        try {
                test.runMethod();
        } catch (AssertionFailure e) {
                addFailure(test, e);
        } catch (Throwable e) {
                addError(test, e);
        }
}
```

Listing 3.6: Wrapping of each test case execution in instance class *ResultCollector*

Once a test object is wrapped into an instance of class *ResultCollector*, it is again wrapped into test fixtures *setUp() and tearDown()* in the method *runMethod()* of test class. After wrapping process, instance of class *ResultCollector* performs monitoring activities over each test case execution and keeps record of results drawn from each test method execution.

3.2.7 Failures and Errors Handling

When actual test method code executes, it calls assertion method of class *Assertion* to assert the result, each such a method in class *Assertion* must take some value(es) to evaluate that it is equal to expected. If expected and actual values are equal then it does nothing but if they are not equal, a method fail is called as mentioned in listing 3.7.

```
public static void fail(String msg) {
        throw new AssertionFailure(msg);
}
```

Listing 3.7: Throw of an exception if a test case is failure

This method takes some messages to guide about what is wrong going on in the given test method. The class *AssertionFailure* is another class which is inherited from chain of classes of which root class is *Throwable*. So, in case of failure, an instance of class *AssertionFailure* is thrown as an exception. These exceptions are finally caught in the *run(GeneralTest test)* method of class *ResultCollector*. In case of failure or error in the test method, each exception is wrapped into an object of class *TestFailureHandler*. The class *ResultCollector* discriminates failure and errors by storing instances of class *TestFailureHandler* in different lists.

3.3 Testing Procedure

As we know, there is a need to transform *JUnit3.8* source code into non-reflective source code. This primary need makes *STF* implementation splits into two steps.

3.3.1 Code Generation

Static Testing Framework describes code generator to generate the test classes according to the implemented design of execution environment. It is not a trivial part of this tool. It is necessary to replace the reflective behavior with static behavior of *Static Testing Framework*. The transformation of *JUnit3.8* source code takes advantage of *Recoder0.94c* provides meta-models of Java projects to manipulate the structure of test classes by adding some more information.

3.3.2 Test Method Execution

It describes static platform for test methods execution , it uses generated code. It describes static creation of test cases in which each test object is associated with a particular test method. After creation of test cases, each test case is executed wrapped by the result object. It handles errors and failures as exception and these exceptions are drawn later in test report.

3.4 Code Generation and Data Collection Methodology

The most difficult part of *STF* implementation is the code generation. The code generator needs to generate code according to design of execution environment. The generated code should be error prone and compatible with the design of execution environment. The code generator is having great scope over the *JUnit3.8* source code. It is performing many monitoring activities to manipulate with *JUnit3.8* source code.

3.4.1 Setting Paths

Recoder0.94c must have information about the projects to which it is going to deal. In order to make *Recoder0.94c* capable to deal with different projects, testers need to set the input paths to all such projects. These input paths are also said to be project settings for class *CrossReferenceServiceConfiguration*. If project paths are not set as input path, *Recoder0.94c* throws exceptions class *UnresolvedClassReferenceType*.

3.4.2 Getting Java Files

Testers also need to set the path for the package containing test classes related to particular project. The code generator takes all the Java files from the specified path. It has a function to filter Java files. In the *Recoder0.94c Library*, class *SourceFileRepository* is like a database that holds information about the project needs to be dealt. The class *SourceFileRepository* makes compilation units of filtered Java files. A compilation unit is also said to be ASTs. The compilation units are held by the class *SourceFileRepository*. Once compilation units are ready, it tells us that project source code is ready to access and manipulate.

3.4.3 File Selection Process and Type Hierarchy

There is an important aspect to select all the files which are subtype of class *TestCase* and class *TestSuite*. They need to regenerate and all such subtypes refer to test classes. A good code generator should also be aware about all the subtypes of class *TestCase* and class *TestSuite*. *STF* code generator has leverage to find all such types.

3.4.4 Screening Each Type

Once code generator gets complete information about the test classes, each test class or type is ready to generate. Each type is generated after passing through bunch of methods and conditions.

a) Error Findings on Code Generation Time

STF design divides its execution in two parts. Some of the errors are caught during code generation and some during test objects execution. The generation of each type depends on some conditions like each type must have constructor such as

public <typename>() {} or public <typename>(String name) {super(name);}, and type modifier should be public. If a type does not meet the above conditions, it is not generated and code generator displays error report.

b) Test Methods Hunting

Each type has property to access its all test methods. The code generator filters all the test methods and put them in a list. A test method must meet the following criteria:

- return type should be void
- modifier should be public

- with no parameter

The purpose of building test method list is to just give awareness to each test class about its test methods need to be executed. It enables a type to access the name of own test methods and to select a particular test method for execution.

c) Building Syntax of Each Type

This is the stage where code generator starts to play with strings. Each type needs to be generated with acceptable syntax. The compilation unit of each type helps to access the source code and to manipulate it.

- **Class Template**

  A class template describes a frame for each type. It defines boundaries for source code need to be generated. Import library is out of the scope of class template. Once some code is generated into the template, a new template is returned.

- **Screening Import Library:**

  As we know, every Java class has some import library that belongs to built in functionality used in the class. When a class is generated, the code generator decides which library need to import. It avoids *JUnit3.8* library and all the classes which are test classes importing from the *JUnit3.8* test project. Once code generator gets complete source of import library for each type, it is screened out to avoid *JUnit3.8* library and meta class references related to test classes. Finally, the screened import library is added to generated class.

- **Formatting Class Declaration**

  Here code generator has complete source code of passed type and it performs some light string operations to just replace some words. The string operations depend on super type of current type. A super type may be a class *TestCase* or class *TestSuite*.

- **Formatting Constructor Declaration**

  Every piece of code for a type can be accessed because it resides in its child tree and child tree can be traversed easily. A constructor with no parameter and a constructor with string parameter are treated differently. When source code of each element is obtained, it is not in proper format, code generator fixes its format. Each method and constructor is passed through a process in which instances of class *MetaClassReference* are collected. A list of class *MetaClassReference* instances is helpful for the import library screening.

- **Formatting Method Declaration**

  Each method in the type requires some necessary manipulations. The code generator works with different functions for different type of manipulations. It makes code generation flexible, error prone and compatible with *STF* implemented design. Most of the manipulations are related to replace some particular words recognized by *STF*. A method may have class declaration that can be accessed in its child tree.

- **Formatting Field Declaration**

  A field is treated as fragment of type or class. A field source code may have some conflicted words that need to be replaced. The code generator provides universal formatting functionality that is not specific to particular element.

- **Inserting Additional Source Code**

  Each type requires a systematic functionality to build the test object packages or suites and then execute the test objects in a remarkable execution hierarchy. This additional code makes a test class little bit intelligent about its test domain. The additional code is also inserted on some important conditions. It may change, if a class is abstract or on behalf of constructor source code.

3.4.5 Code Generation Report

Each generated type refers to a Java file. Once all the types are generated, code generator shows a pretty nice report of successfully generated test classes and it also notifies error if they occur.

3.5 General Differences between JUnit3.8 and STF

Static Testing Framework finds many prospective to differentiate it from *JUnit3.8 Framework.*

3.5.1 Textual Representation of Results

JUnit3.8 Framework gives results in graphical representation whereas static framework represents results in textual form. The resulting parameter contains complete information whatever the exception is pointing to an error or failure. These results are representing on the Java console. It also gives information about the qualified tests and it also gives complete trace of any error in the test method or test failure.

3.5.2 Setting of Necessary Paths

Users must have to set the properties of class *CrossReferenceServiceConfiguration* for the following paths:

a) recoder.jar
b) junit.jar
c) classes need to be tested

3.5.3 Deletion of Generated Classes

Users need to delete generated test classes each time for each new testing. New test classes can be generated either in the same package in which execution environment of *Static Testing Framework* works or in separate package. In both cases, users need to delete the generated classes. If a user does not do so, it will cause of complexity.

3.5.4 Multiple Testing Project Problems

If there are more than one testing projects in the same Java project, users need to set path for the package in which required test classes are existing as shown in listing 3.8. There is one benefit of setting the path. It can optimize performance of code generator and reduce the load of all entities inherited from class *TestCase*.

```
String testClassPath = "D:/Practical/Thesis/src/factorial/";
ArrayList<File> testClassFiles = filterJavaFiles(testClassPath);
String[] testClassAbsolutePath = new ring[testClassFiles.size()];
for (int jIndex = 0; jIndex < testClassFiles.size(); jIndex++) {
        testClassAbsolutePath[jIndex] =testClassFiles.get(jIndex)
                .getAbsolutePath();
}
sfr.getCompilationUnitsFromFiles(testClassAbsolutePath);
```

Listing 3.8: Path setup and selection of Java files for *JUnit3.8* Source Code

3.5.5 Error Handling at Code Generation Time

Some of the errors are handled at the time of code generation rather than at run time. The *Code Generation Time Errors* are as below:

a. *<class name> has no public constructor <class name>(String) or <class name>()*
b. *<class name> is not public*
c. *<test method> is not public*

3.5.6 Instances of Test Classes as Parameter

During the test pack building process for the final execution, users need to pass object of test class in *addTestPack(new <class name>())* method rather then class itself. The purpose of passing instance of test class is to create its further instances statically. *Static Testing Framework* does not allow test class to get its instances dynamically.

3.5.7 Design of STF

In *JUnit3.8 Framework*, class *TestCase* is behaving like a middle layer between *Test* interface and test classes. The class *TestCase* class inherits from class *Assert* and all the tests are passed through this class during execution. The execution environment of *Static Testing Framework* implemented different design technique, there is no middle layer. Each test class is interacting with interface directly by implementing it. It is also getting all the properties of class *Assertion* through inheritance.

Chapter 4
Evaluation

This chapter describes progress, efficiency and effectiveness of implemented tool. Mainly, it focuses on success factor that tells readers what are the set goals and what are achieved goals. The purpose of evaluation process is to record and organize results and then comparing them with expected results. The evaluation process draws different results and these statistical results are helpful to measure the efficiency and to extend the implementation for future work.

4.1 Evaluated Projects

There is a prime need to evaluate some large scale projects providing several number of test classes with different style of written test cases. Following projects are evaluated on the below specified machine.

Operating System	: Microsoft Windows XP Professional (5.1, Build 2600)
System Manufacturer	: Hewlett-Packard
System Model	: HP Compaq nc6000 (DJ256A#AK8)
Processor	: Intel(R) Pentium(R) M processor 1600MHz
Memory	: 768MB RAM
Java	: jdk1.6.0_14

4.1.1 Recoder0.94c Library

It is built on *Java Framework.* It provides a platform for meta-programming of *Java Source Code*. The main purpose of *Recoder0.94c Library* is to analyze the source code. It has also ability to transform and modify source code. I am using *Recoder0.94c*.

a) Statistical Measures

I changed class *recoder.testsuite.completeCoverage.CompleteCoverage* by extending it from class *TestCase*, otherwise it can not be regenerated. The reason is that class *CompleteCoverage* has a method *suite()* and this method is building suites for *recoder.testsuite.completeCoverage.NameInfoCoverage* and *recoder.testsuite.completeCoverage.KitCoverage* "by passing class as parameter". According to implementation, these class references need to convert in "instances of *NameInfoCoverage* and *KitCoverage* as parameter" because STF allows only instances of test classes. If class *CompleteCoverage* is not inheriting from class *TestCase*, it means it will not be transformed and if it will not transform into new generated class then class references will not be converted to "instances of *NameInfoCoverage* and *KitCoverage* as parameter".

JUnit3.8 Results		**STF Results**	
Tests Found	19	Tests Found	19
Failures	2	Failures	2
Errors	2	Errors	4

Table 4.1: Statistical Measures of *Recoder0.94c* Library

b) Bugs Findings and Why Different Results Are?

- STF found two more errors as compared to tests read by *JUnit3.8*, the details of this problem describes that:
  i. In class *recoder.testsuite.transformation.TransformationTests*, there is a method named *testReadOnly*. This method uses a string *"Test"* which is replaced by *"GeneralTest"* during code generation. In fact, it should not be replaced. The reason is that *STF* has an interface named *GeneratTest* performing possibly same responsibilities as *JUnit3.8* interface *Test*. However *STF* replaces all such strings *"Test"* with *"GeneralTest"*. In this particular case as shown in listing 4.1, there is a class *Test* in the test package of *Recoder0.94c*. The code as shown in listing 4.1 is actually accessing that class but when the name of this class is replaced by *GeneralTest*, there is an error because a class with the name of *GeneralTest* does not exist in the test package of *Recoder0.94c*.

In method *testReadOnly(),* *sc.getCrossReferenceSourceInfo().getReferences(sc.getNameInfo( ).getType("Test"));* should not be replaced by *sc.getCrossReferenceSourceInfo().getReferences(sc.getNameInfo( ).getType("GeneralTest"));*

Listing 4.1: An error occurrence due to replacement of a string "*Test*"

  ii. The method *testBug2013315()* in class *recoder.testsuite.fixedbugs.FixedBugs.class* having same problem as described in section 4.1.1 (b) (i). The code is mentioned below:

String cuText = "class X {{Test instance = new Test();" + *"instance.myMap.get(\"\").shortValue(); }}";* *Should not be replaced by* *String cuText =* *"class X {{GeneralTest instance = new GeneralTest();" +* *"instance.myMap.get(\"\").shortValue(); }}";*

Listing 4.2: An error occurrence due to replacement of a string "*Test*"

- In class *recoder.testsuite.semantics.SemanticsTest*, other types class *recoder.testsuite.semantics.MethodTest* and class *recoder.testsuite.semantics.NameTest* are used, class *SemanticsTest*, class *NameTest* and class *MethodTest* reside in the same package. Due to same package, class *NameTest* and class *MethodTest* is not imported. So, when class *SemanticsTest* is generated, class *NameTest* and class *MethodTest* are not included in the import list. It gives error wherever it is used and it has to be imported after code generation. The one possible solution is to generate all the classes presented in the test package of given Java project. A class that is not a subtype of *TestCase* or *TestSuite* should be generated with the same source code without having additional functionality.

- In class *staticframework.ParallelTest*, testers need to remove import *recoder.testsuite.CompleteTestSuite*, generated class *CompleteTestSuite* should be used. During code generation, code generates screens import library. It removes all *JUnit3.8 Framework* classes from list of import classes and all the classes which are used as class reference in the test class. Therefore, class *CompleteTestSuite* in the class *ParallelTest* neither belong to *JUnit3.8 Framework* nor used as class reference in the class *ParallelTest*. So after generation of *ParallelTest*, the imported class *CompleteTestSuite* refers to class *recoder.testsuite.CompleteTestSuite* from *JUnit3.8* source rather than generated class *CompleteTestSuite*.

4.1.2 Grail Library

It describes a tool that can be used to construct and manipulate a graph. The manipulation of graph takes place due to different algorithms. This library is useful for software visualization.

a) Statistical Measures

I built one of the test classes myself named class *grailtest.CompleteGrailTest* to endorse all the test classes in a single suite. There is no class in *Grail Test Suite* representing comprehensive test of *Grail Library*.

JUnit3.8 Results		STF Results	
Tests Found	90	Tests Found	90
Failures	3	Failures	2
Errors	0	Errors	3

Table 4.2: Statistical Measures of *Grail* Library

b) Bugs Findings and Why Different Results Are?

STF found only two failures because *STF* found an error in a test method which is failure read by *JUnit3.8*. The detail of found errors is as under:

- An error exception *NullPointerException* throws in *grailtest.TestContains.testContains*, *grailtest.TestContains.testEmptyGraph* and *grailtest.TestProperties.testEmptyGraph*. Data members are being initialized by calling the test fixture *setUp()*. In case of *STF* implementation, there is no such a class like *TestCase*. In generated code, the method *runMethod()* in each test class calls *setUp()* and *tearDown()* method as shown in listing 4.3.

```
public void runMethod() throws Throwable {
        Throwable excep = null;
        setUp();
        try {
                runTestMethod(this);
        }
        catch(Throwable runTestExcep){
                excep = runTestExcep;
        throw excep;
        }
        finally {
                try {
                        tearDown();
                } catch (Throwable tearDownExcep) {
                if (excep == null)
                        excep= tearDownExcep;
                }
        }
        if (excep != null)
                throw excep;
}
```

Listing 4.3: Shows limited scope of test fixtures execution

Therefore, *runMethod()* as shown in listing 4.4 is associated with the created test case. So when test case *"test1"* is executed the method *runMethod()* is called which has no calls to *setUp()* and *tearDown()*. In case of implemented approach, test fixture *setUp()* is not executed when a test case is created in the test class.

```
static TestProperties test1 = new TestProperties("Empty Graph
Properties"){
        public void runMethod() {
                testEmptyGraph();
        }
};
```

Listing 4.4: A representation of test case creation

During the building of suite as shown in listing 4.5, when a test case *"test1"* is added to the list of test cases, this test case always refers to *runMethod()* which is associated with it during creation and this type of method call is not containing calls for test fixtures.

```
public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(test1);
        suite.addTest(test2);
        return suite;
}
```

Listing 4.5: Building of test suite containing multiple test cases

In case of JUnit3.8, the test fixture *setUp()* is always executed before the execution of each test method and test fixture tearDown() is always executed after the execution of each test method as shown in listing 4.6 and as described in section 2.1.3 (b). The possible solution to avoid this problem is to introduce a class like *TestCase* to execute the test fixtures any way.

```
public void runBare() throws Throwable {
        Throwable exception= null;
        setUp();
        try {
                runTest();
        } catch (Throwable running) {
                exception= running;
        }
        finally {
                try {
                        tearDown();
                } catch (Throwable tearingDown) {
                        if (exception == null) exception=
                        tearingDown;
                }
        }
        if (exception != null) throw exception;
}
```

Listing 4.6: Scope of test fixtures execution in *JUnit3.8*

4.1.3 Antlr3.1.1 Library

Antlr3.1.1 (ANother Tool for Language Recognition) is a language tool for constructing compilers, interpreters, and translators on the basis of described grammar. *Antlr3.1.1* provides an interface to describe grammar with facility of guidelines and editing tools. It can construct and represent *ASTs* for the described grammar.

a) Why am I using antlr3.1.1.v?

The *Antlr3.1.1*version is built on *JUnit3.8*. The later versions followed by *Antlr3.1.1* are built on *JUnit4.0*.

b) Statistical Measures

I built one of test classes myself named class *org.antlr.test.CompleteANTLRTest* to endorse all the test classes in a single suite. There is no class in *Antlr3.1.1* test suite representing comprehensive test of *Antlr3.1.1* library.

JUnit3.8 Results		STF Results	
Tests Found	10	Tests Found	10
Failures	75	Failures	58
Errors	0	Errors	0

Table 4.3: Statistical Measures of *Antlr3.1.1* Library

c) Bugs Findings and Why Different Results Are?

- The difference in number of failure tests is just because of class *org.antlr.test.DebugTestAutoAST* which inherits all the test methods from *org.antlr.test.TestAutoAST* and class *org.antlr.test.DebugTestRewriteAST* inherits all the test methods from class *org.antlr.test.TestRewriteAST*, the constructor with no parameters of class *DebugTestAutoAST* sets the value of debug like ***public*** *DebugTestAutoAST() {debug* = ***true****;}*.

  In case of *STF*, there is no call to constructor with zero parameters which sets the *debug* as *true* and the constructor with parameter *String* does not set *debug* as *true*. So the value of debug is left unset. Each generated type must have constructor with one parameter type of string to specify the name of test method. In *JUnit3.8 Framework*, each test method name is accessed using reflection, constructor with string type are created at run time and then test object is created using recently created constructor by assigning name of test method. So static execution of test objects never calls constructor with no parameters and the values in the constructor are left unset.
- *STF* implementation has leverage to import all the classes in the generated code need to execute test classes uniformly. If a test class uses another class or its functionality that is not imported but exists in that package then testers need to put such a class in the package of generated test classes. We take an example of class *org.antlr.test.BaseTest*, another class *org.antlr.test.ErrorQueue* is used in this fashion.

4.1.4 Software Technology Project

It is small application for graph construction and manipulation. It provides an infrastructure to construct a graph. There are different well-known algorithms related to graph theory can be applied on constructed graph. It has also privilege to write *Graph Markup Language* that can be readable by a graph editor.

a) Statistical Measures

This project is working well and there are no bugs found during its execution.

JUnit3.8 Results		STF Results	
Tests Found	13	Tests Found	13
Failures	0	Failures	0
Errors	0	Errors	0

Table 4.4: Statistical Measures of STP

4.2 Performance of Static Testing Framework

The performance of *STF* is low but the performance is not a prime goal. The main focus of this implementation is on static analysis. *STF* design divides implementation in two parts which causes to make its performance low. As the name indicates, the code generator generates *JUnit3.8* test classes in corresponding files. The files are considered to be Java files (Java classes). The code generation process goes to expand as the size of *JUnit3.8* source code increases. The code generator deals with several string data types to manipulate the extracted data from *JUnit3.8* source code and then writes manipulated data into corresponding files. Once code is generated, each generated class has lot of lines of code due to extra functionality.

Chapter 5
Future Work

This section tells the readers, what more can we expect from *Static Testing Framework*? The achievement of following described expected goals can cause of more efficiency and effectiveness. These expected goals motivate to improve the performance of static framework.

5.1 Interactive Configuration Design

An interactive environment should be designed that will be responsible to configure *JUnit3.8* source code, generated classes and execution of created test cases at once. The design of implemented structure requires a set of paths. Once test classes are generated on a specific location or package, that package needs to refresh to get access to the generated classes. There should be implementation dedicated to handle this problem.

5.2 Development of Plug-in

An eclipse plug in can also be developed to handle complete configuration at once which is working on manual setting right now. The developed plug in will provide GUI mode to configure all necessary requirement needed by static structure. There would be some facility to set paths for the *JUnit3.8* and *Recoder0.94c*. It is also required specific implementation. This specific implementation will give access to the generated classes and will provide support for code generation and execution of test cases at once.

5.3 Graphical Representation of STF Results

The results are shown in simple textual form but it would be possible to show them in a graphical form by discriminating each and every thing held by result object. It can make this tool friendlier.

5.4 JUnit4 Support

The design of *JUnit4* is relatively different as compared to *JUnit3.8*. The class *TestCase* is no longer working in *JUnit4 Framework*. Each test class can be defined independently and there is no need to make each test class as subtype of class *TestCase*. Now class *TestRunner* is more intelligent to find out test methods but it is very necessary to mention *@Test* annotation for each test method. *STF* shall be extended to *JUnit4 Framework*.

5.5 Removal of Unwanted Library

STF concerns screening of import library for only class references used within the test class and classes from *JUnit3.8 Framework* but if a class is working and it is not used as class reference within the program, it will not be removed from the import library and produces conflict between generated test class and *JUnit3.8* test class. All such type of classes should be removed from the import library. This problem is raised during evaluation of *Recoder0.94c* library and *Antlr3.1.1* in section 4.1.1 & section 4.1.3.

5.6 Execution of Test Fixtures

The test fixture *setUp()* must be called before execution of each test method and *tearDown()* must be called after execution of each test method. A tester can write test cases in such a way in which data members are initialized in *setUp()* methods. If a test case is created in the test class and a particular test method is associated with test object then test case must call method *runMethod()* of created test case without calling test fixtures. This problem is discussed in evaluation of *Grail* library in section 4.1.2.

5.7 Generation of Constructor as Per Real Definition

STF must generate constructor with no parameters and constructor with string parameter for each test class. They should be generated according to real definition of test class. It requires more work at code generation time but it will make static structure more easy. It can fix some errors described in evaluation of *Antlr3.1.1* library in section 4.1.3.

5.8 Generation of Import Library Used By Test Classes

If a *JUnit3.8* test class is accessing functionality from one or multiple classes which are not subtype of *TestCase* or *TestSuite* residing in the same testing package. Off course, these types of classes will not be included in the import library of test class. When this test class is generated, it always looks for one or multiple classes of which functionality it is using. So, these type of classes in the Java test project which have no relation with *TestCase* or *TestSuite* should also be generated with same source code without having extra functionality. The problem is mentioned in evaluation of *Recoder0.94c* library in section 4.1.1.

5.9 Omit Additional Working to JUnit3.8 Source

If a class in a given Java test project, builds a test suite by taking test classes as class references as shown in listing 5.1. In order to convert class references to the instances of test classes, testers need to extend this class from *TestCase*. These types of classes building only test suites should be included in the generated test domain without doing any change to the *JUnit3.8* source.

```
public class CompleteCoverage {
        public static Test suite() {
                TestSuite suite = new TestSuite(
                "Test for recoder.testsuite.completeCoverage");
                suite.addTestSuite(NameInfoCoverage.class);
                suite.addTestSuite(KitCoverage.class);
                return suite;
        }
}
```

Listing 5.1: A class building test cases of test classes

Chapter 6
Related Work

This chapter describes possible solution to resolve the reflection. Reflection enables a program to access and modify itself during execution. The analysis to resolve the reflective calls consists of the following three steps [16]:

a) A static technique is deployed that refers to context-insensitive points-to analysis. It is used to determine all the constant strings or strings from the external resources in a program representing class names. The reflective calls can be resolved if all such strings refer to possible source. The points in a program where input strings are required provided by a user or external resources are said to be specification points.

b) The specification points in a program can be large. Reflection resolution can take advantage of casting whenever available despite to specify every possible input string. It can be concluded that newly created object must refer to the type used in the cast as shown in listing 6.1.

```
String className = “”;
Class clazz = Class.forName(className);
Object obj = clazz.newInstance();
TestClass tObj = (TestClass) o;
```

Listing 6.1: Typical use of reflection to create new objects

c) A program may have specification points where cast does not take place. A class name can be provided through external resources instead of constant strings. It is necessary to specify all such points in order to obtain conservative approximation of call graph.

6.1 Call Graph Discovery

A call graph defines relation between a caller and callee. A static technique to determine the reflective calls is integrated with context-insensitive point-to analysis to discover call graph. The purpose of points-to analysis is to determine points of variables and type information of these points. These points information is used to resolve the potential targets. The analysis of reflective calls causes further expansion of call graph and this expanded call graph is used for further analysis of point-to relations. In order to discover the call graph, a call graph discovery algorithm is used which follows context-insensitive points-to analysis technique.

6.2 Reflection Resolution

The points-to analysis just discovers reflective calls, it can not resolve the reflective calls like *clazz.newInstance()*. The points-to analysis tracks method, field and constructor objects. Points-to information is used to determine the targets of *clazz.newInstance()* and add calls to the call graph.

6.2.1 BDD Program Database

A *Binary Decision Diagrams* program refers to set of rules which express a particular program domain. A *BDD* algorithm expresses program representation and points-to analysis results in form of *Datalog*, a logic programming. For example, there is a *Datalog* relation *Red(X):- Blue(Y), Green(Z)* says that *Red(X)* is true if *Blue(Y)* and *Green(Z)* are all true. These *Datalog* relations reside in a database which is called *BDD* program database. Following are *Datalog* relations represent the input program and points-to results.

actual(i, z, v) means that variable *v* is *zth* argument of the method call at *i*
ret(i, v) means that variable *v* is the return result of the method call at *i*
assign(v1, v2) means that there is an implicit or explicit assignment
load(v1, f, v2) means that there is a load statement *v2 = v1.f* in the program
store(v1, f, v2) means that there is a store statement *v1.f = v2* in the program
string2class(s, t) means that string cons *s* is the representation of the name of type *t*
calls(i,m) means that invocation site *i* may invoke method m
Points-to results are represented with the relation *vP*:
vP(v, h) means that variable v may point to heap object *h*

6.2.2 Reflection Resolution Algorithm

Reflection resolution algorithm is a program that can interpret *Datalog* relations and it is connected with call graph in order to examine *Datalog* relations. To find the reflection targets such as *Class.forName* and *Class.newInstance*, *BDD* defines data log rules. The relation *classObject(i,t)* contains pairs of *(i,t)* where *i* belongs to set of invocation site *I (i ∈ I)* and *t* belongs to set of return type *T (t ∈ T)*. It states that invocation site *I* returns an object of type *t* if there is an edge from *i* to *Class.forName* in the call graph as shown in listing 6.2.

```
classObjects(i, t) : –  calls(i, "Class.forName"),
                        actual(i, 1, v), vP(v, s), string2class(s, t).
```

Listing 6.2: Return an Object of Type *t*

The relation *newInstanceTarget(i,t)* contains pairs of *(i,t)* where *i* belongs to set of invocation site *I (i ∈ I)* and *t* belongs to set of return type *T (t ∈ T)*. It states that invocation site *i* returns new object of type *t* if call graph contains an edge from *i* to *Class.newInstance* as shown in listing 6.3.

```
newInstanceTargets(i, t) : – calls(i, "Class.newInstance"),
                             actual(i, 0, v), vP(v, c),
                             vP(vc, c), ret(ic, vc),
                             classObjects(ic, t).
```

Listing 6.3: Return New Instance of Type *t*

6.2.3 Resolution of Specification Points

Points-to analysis allows us to determine the provenance of a string. A provenance may be a system property or configuration of a file which is used to compute the specification points. The provenance of a string can be computed by propagating *Datalog* relations like assign, store, load and calls. A backward propagation is used for

each input string through string concatenation operations which makes specification points as close as possible to input strings.

Chapter 7
Conclusion

This chapter draws results obtained from the implementation of *STF*. It is focusing on the progress of research and describing future work in the light of perspective goals.

7.1 Summary

The aim of this research is to contribute in static analysis of *Java Source Code*, building process and execution of test cases. *Static Testing Framework* provides a platform to perform many activities statically. As described in chapter 4, *STF* comes with possible same functionality as provided by *JUnit3.8 Framework* but with different technique and this unique technique makes it convenient for static analysis. *Static Testing Framework* suggests possible solutions for dynamic behavior without using reflection. It draws almost same results as *JUnit3.8 Framework* by modifying test classes and then running them.

7.2 Conclusion

It can perform following prospective tasks in context of static structure:

Static Testing Framework generates test classes and adds more functionality. The new functionality is responsible to make a static structure convenient for building and execution of test pack. It provides data members and member functions to give awareness to the test classes about its test methods. For each new testing, a user needs to delete generated test classes. It does not affect performance of *STF* execution environment but makes it more complex. Consequently, generated test classes are comprehensive and robust.

When a test pack is built by passing instance of test class, instances of test class (test objects) are created statically. The number of test objects depends on the number of test methods found in the corresponding test class. Each generated class needs to be tested is dealt same way. Each test class has its own test objects residing in a vector list. After successful creation of test objects, test methods are executed statically.

Therefore, *STF* uses *Recoder0.94c Library* and this library gives facility of meta-programming. The meta-programming can be used to avoid reflection and enables *STF* to build a static structure that emulates possible functionality as reflection does. All the *Code Generation Time Errors* are notified at the time of code generation. *Static Testing Framework* comprises two phases of testing, one is responsible for generating *JUnit3.8* source code and the other performs execution of created test cases.

References

[1] M. Burke & M. Coyner, 2003. JUnit. In M. Burke, ed. *Java™ Extreme Programming Cookbook*. USA: O'Reilly & Associates, Inc., 2003, pp. 61 – 102.

[2] V. Messol & T. Husted, 2004. Understanding Unit Testing Frameworks. In V. Messol, ed. *JUnit in Action*. USA: Manning Publications Co., 2004, pp. 10.

[3] V. Messol & T. Husted, 2004. Exploring JUnit. In V. Messol, ed. *JUnit in Action*. USA: Manning Publications Co., 2004, pp. 17 – 37.

[4] J.B. Rainsberger & S. Stirling, 2005. Organizing and Building JUnit Tests. In S. Scott, ed. *JUnit Recipies*. USA: Manning Publications Co., 2005, pp. 71 – 99.

[5] J.B. Rainsberger & S. Stirling, 2005. Managing Test Suites. In S. Scott, ed. *JUnit Recipies*. USA: Manning Publications Co., 2005, pp. 102 – 127.

[6] J.B. Rainsberger & S. Stirling, 2005. Running JUnit Tests. In S. Scott, ed. *JUnit Recipies*. USA: Manning Publications Co., 2005, pp. 173 – 185.

[7] P. Baker, Z. Ru Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker & C. Williams, 2007. Test Execution with JUnit. In C. Williams, ed. *Model Driven Testing*. Berlin: Springer, 2007, pp. 149 – 156.

[8] C. Beust & H. Suleiman, 2007. Testing Design Patterns. In H. Suleiman, ed. *Next Generation Java Testing*. Massachusetts: Pearson Education, Inc., 2007, pp. 23 – 150.

[9] IT-Consultant Marco van Meegen. *Slime UML*. [Online]
Available at: http://www.mvmsoft.de/content/plugins/slime/index.htm
[Accessed: 15-11-2009].

[10] *JUnit Cook's Tour*. [Internet]
Available at: http://junit.sourceforge.net/doc/cookstour/cookstour.htm
[Accessed: 01-09-2009].

[11] *JUnit 3.8 Documented Using Collaborations.*[Internet]
Available at: http://portal.acm.org/citation.cfm?id=1350812
[Accessed: 05-09-2009].

[12] *RECODER Introduction and Program Model.*[Internet]
Available at: http://www.info.uni-karlsruhe.de/~compost/CurrentCOMPOST/RECODER/doc/manual.html [Accessed: 22-09-2009]

[13] *RECODER Features and Possibilities.*[Internet]
Available at: http://www.info.uni-karlsruhe.de/~compost/CurrentCOMPOST/RECODER/index.html
[Accessed: 22-09-2009].

[14] *Meta-programming.*[Internet]
Available at: http://en.wikipedia.org/wiki/Metaprogramming
[Accessed: 07-10-2009].

[15] *Java Reflection Overview.*[Internet]
Available at: http://en.wikibooks.org/wiki/JavaProgramming/Reflection/Dynamic_Invocation
[Accessed: 07-09-2009].

[16] B. Livshits, J. Whaley, M.S. Lam, 2005. *Reflection Analysis for Java.*
[Online] Stanford, USA: Stanford University (Published 2005)
Available at: suif.stanford.edu/papers/aplas05r.pdf [Accessed: 09-09-2009].

Linnæus University
School of Computer Science, Physics and Mathematics

SE-351 95 Växjö / SE-391 82 Kalmar
Tel +46-772-28 80 00
dfm@lnu.se
Lnu.se