GenUTest: An Automatic Unit Test & Mock Aspect Generation Tool

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree in Tel-Aviv University, School of Computer Science

by

Benny Pasternak

Prepared under the supervision of Dr. Shmuel Tyszberowicz, and Prof. Amiram Yehudai

January, 2008
Acknowledgments
Abstract

Unit testing plays a major role in the software development process. What started as an ad hoc approach is becoming a common practice among developers. It enables the immediate detection of bugs introduced into a unit whenever code changes occur. Hence, unit tests provide a safety net of regression tests and validation tests which encourage developers to refactor existing code with greater confidence.

Agile development methodologies, such as XP and TDD, strive to shorten development cycles and constantly release working software. Unit testing is one of the major corner stones of such methodologies. It is required that all the software classes have unit tests, and that they can be executed using an automated unit testing framework. Violation of this requirement brings the development process to a complete stop.

However, not all software systems contain unit tests. When changes to such software are needed, writing unit tests from scratch, which is hard and tedious, might not be cost effective.

In this thesis we propose a technique which automatically generates unit tests for software that does not have such tests. We have implemented GenUTest, a prototype tool which captures and logs inter-object interactions occurring during the execution of Java programs, using the aspect oriented language AspectJ. These interactions are used to generate JUnit tests. They also serve in generating mock aspects – mock object like entities, which enable testing units in isolation.

The generated JUnit tests and mock aspects are independent of the tool, and can be used by developers to perform unit tests on the software. Moreover, the comprehensiveness of the unit tests depends on the software execution. These executions can be planned by developers to obtain high code coverage and in turn generate unit tests with similar code coverage. To validate this belief we performed some experimentation with the tool and applied it on several open source projects such as NanoXML and JODE. In these experiments we compared the code coverage obtained by synthetic ‘system’ tests to the code coverage obtained by the generated unit tests. The preliminary results show that obtaining similar code coverage is feasible. Hence, a wide unit testing suite can be formed which in turn can enable the continuation of the project using agile development methodologies for the first time.
5.2 Mock Object Frameworks .................................................. 27
5.3 Mock Aspects ............................................................. 28

6 Conclusion ................................................................. 33
   6.1 Experimentation ...................................................... 33
   6.2 Capabilities and Limitations ....................................... 34
   6.3 Future Work .......................................................... 35
List of Figures

1.1 A typical system unit dependency graph ........................................ 2
1.2 GenUTest architecture. ................................................................. 4
1.3 The classic calculator example. ..................................................... 7
1.4 A test for the add method. ............................................................. 7
1.5 A test that checks that an exception is thrown when dividing by zero. .... 8
1.6 Unit test results reported by JUnit. ................................................ 8

3.1 A sequence diagram describing a scenario of the stack behavior. ........... 13
3.2 An XML representation of a Method-Call event. .............................. 16

4.1 Unit test generated for pop() method-call. .................................... 21
4.2 Method-calls invoked on the objects obj1, obj2, and obj3. ................. 24
4.3 Statements generated to restore the correct state of obj1 and obj2. ....... 24
4.4 Unit test generated for exception throwing pop() method-call. .......... 25

5.1 Object creation using the factory pattern. ...................................... 27
5.2 Testing pop with the use of the EasyMock framework. ....................... 29
5.3 A Class Under Test (CUT) and its incoming and outgoing method-calls. . 30
5.4 Unit test for pop() method-call with helper statements. ................... 31
5.5 A code snippet of the mock aspect generated for IntStack. .................. 32
Chapter 1

Introduction

Unit testing plays a major role in the software development process. A unit is the smallest testable part of an application; in the object oriented paradigm it is a class. A unit test consists of a fixed sequence of method invocations with fixed arguments. It explores a particular aspect of the behavior of the Class Under Test, hereafter CUT. Testing a unit in isolation is one of the most important principles of unit testing. However, the CUT usually depends on other classes, which might even not exist yet. Mock objects [12] are used to solve this problem by helping the developer break those dependencies during testing, thus testing the unit in isolation.

Some of the benefits of unit testing are:

1. **Facilitates Change**: Unit testing enables programmers to safely refactor code, and make sure it works. By writing test cases for all methods whenever a change causes a regression bug, it can quickly be identified and fixed. This encourages programmers to change the code whenever refactoring can provide a better program structure.

2. **Simplifies Integration**: The integration involves units that have already been tested, thus easing the process.

3. **Documentation**: Unit tests document the use of the units, thus providing an example to programmers on how to correctly use a particular unit.

Writing unit tests is a hard and time consuming task. The level of difficulty of a unit test can be determined by two independent characteristics of the CUT:

1. The number of units it depends on.

2. The complexity of its state.

Figure 1.1 shows a typical dependency graph of units in the system. The circle shaped nodes represent units that do not depend on other units in the system. Testing such units is usually quite simple. The rectangle shaped nodes represent units that depend on other units in the system, and thus are harder to test. Testing units that have states requires some setup code that enables to bring the unit to the desired state before running a particular test case. Testing units that depend on other units in the system require the use of mock objects in order to test the unit in true isolation. Mock object is a unit testing pattern [11] that is classified under the
Mock objects are used to simulate or mock the behavior of a real unit. Mock objects respond to method-calls in the same manner as the real units would have responded. However, mock objects do not perform any action and immediately return to the caller.

1.1 Motivation

Extreme Programming (XP) [31] adopts an approach that requires that all the software classes have unit tests; code without unit tests may not be released. Whenever code changes introduce a bug into a unit, it is immediately detected. Hence, unit tests provide a safety net of regression tests and validation tests. This encourages developers to refactor working code, i.e., change its internal structure without altering the external behavior [6].

The number of unit tests for a given project might be very large. For instance, Microsoft reported that the code for unit tests is often larger than the code of the project [4]. In order to effectively manage unit tests, execute them frequently, and analyze their results, a unit testing framework should be used [31]. The framework automatically executes all unit tests and reports their results. One of the most popular unit testing frameworks is JUnit [37, 8], which helps to standardize the way unit tests are written and executed.

The cost of the maintenance phase is estimated to comprise at least 50% of the software’s total life cycle [19]. At this phase, the software is already being used, and future versions of the software, which include new features as well as bug fixes, continue to be developed. Unit tests can assist developers during the maintenance phase. Nevertheless, not all developed software
contains unit tests. Writing effective tests is a hard and tedious process, and developing them from scratch at the maintenance phase might not be cost effective. In this case they are usually not written, and maintenance continues to be a difficult process.

We propose a technique which automatically generates unit tests for systems that do not have such tests. This is achieved by capturing and recording the execution of the software in order to obtain test cases. The recorded data can then be used to construct unit tests for the software.

We have implemented GenUTest, a prototype tool, which was first presented in the Haifa Verification Conference 2007 in a paper [16] that is based on this thesis. The tool captures and logs inter-object interactions occurring during the execution of Java programs. The recorded interactions are then used to generate JUnit tests and mock object like entities called mock aspects. These can be used independently by developers to test units in isolation. Moreover, the comprehensiveness of the generated unit tests depends on the software execution. Software executions covering a high percentage of functional requirements are likely to obtain high code coverage and in turn generate unit tests with similar code coverage. Such executions can be planned by the developers with the assistance of the quality assurance personnel, who are responsible for creating test scenarios that exercise the functional requirements of the software and ensure their correctness. Thus, a wide unit testing suite can be formed which can enable the continuation of the project using agile development methodologies for the first time.

Figure 1.2 presents a high level view of GenUTest’s architecture and highlights the steps in each of the three phases of GenUTest: capture phase, generation phase, and test phase. In the capture phase the program is modified to include functionality to capture its execution. When the modified program executes, inter-object interactions are captured and logged. The generation phase utilizes the log to generate unit tests and mock aspects, mock object like entities. In the test phase, the unit tests are used by the developer to test the code of the program.

The interactions are captured by utilizing AspectJ, the most popular Aspect Oriented Programming (AOP) extension for the Java language [25, 10]. The rest of this chapter provides a brief review of the AspectJ language and a quick introduction to JUnit.

1.2 AspectJ

Aspect Oriented Programming (AOP) [9] is a programming paradigm which enables developers to develop software that maintains the principle of Separation Of Concern (SOC), specifically cross-cutting concerns. SOC is the process of breaking a computer program into distinct features that overlap in functionality as little as possible. Both the procedural programming and object-oriented programming paradigms provide constructs that support this process. However, SOC can not be achieved in those two paradigms for cross-cutting concerns; concerns which cut across multiple modules in the program. Logging, which is a strategy that affects every single logged part of a system, is an example of a such a concern. Hence, the logging code is spread over a number of modules. Introducing any change to the logging mechanism may require modifying all affected modules. The AOP paradigm provides the means to handle cross-cutting concerns and to encapsulate them in class like entities called aspects. Aspects contain code that implements cross-cutting concerns and also provide the AOP compiler with information on how
to weave the cross-cutting code with the rest of the program’s code.

AspectJ [25, 10] is an aspect-oriented extension created for the Java programming language. It is available in Eclipse Foundation open-source projects, both stand-alone and integrated into Eclipse [30]. AspectJ has become the widely-used de-facto standard for AOP by emphasizing simplicity and usability for end users.

The Dynamic Join Point Model

A join point model is a critical element in AspectJ that makes it possible to define the dynamic structure of crosscutting concerns. Join points are well-defined points in the execution of the program. A pointcut picks out certain join points and values at those points. Finally, a piece of advice is code that is executed when a join point is reached.

AspectJ supports various kinds of join points. These include: method-call join points, method-execution join points, constructor-call and execution, field set and reference, and more.
A method-call join point encompasses the actions of an object receiving a method-call. It includes all the actions that comprise a method-call, starting after all arguments are evaluated up to and including return (either normally or by throwing an exception).

A method-execution join point differs from a method-call join point by the context of the execution. In the call join point it is the caller which is the executing object while at the execution joint point it is the object where the method is declared.

At runtime, each method-call is a different join point, even if it arrives from the same expression in the program. Many other join points may be active while a method-call join point is executing – all the join points that occur while executing the method body, and in those methods called from the body. We say that these join points execute in the dynamic context of the original call join point.

Pointcuts

Pointcuts match certain join points in the program flow. For example, the pointcut call(void Point.setX(int)) defines the set of all join points that represent calling a method with the signature void Point.setX(int).

A pointcut can be built out of other pointcuts with and, or, and not (spelled &&, ||, and !). Pointcuts can also identify join points from many different types. AspectJ allows programmers to define their own named pointcuts with the pointcut form, and whenever this definition is visible, the programmer can simply use that name to capture this pointcut.

AspectJ also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name. This feature is called property-based crosscutting. The simplest of these involve using wildcards in certain fields of the method signature.

The cflow pointcut identifies join points based on whether they occur in the dynamic context of other join points. cflow(<join point signature>) matches each join point that occurs in the dynamic context of the specified join point. This picks out each join point that occurs between when a method that matches the join point signature is called and when it returns (either normally or by throwing an exception).

Advice

Advice is used to actually implement crosscutting behavior. An advice brings together a pointcut (to match join points) and a body of code (to run at each of the matched join points). When a pointcut matches a certain join point, its advice is performed. AspectJ has several different kinds of advice. The before advice runs as a join point is reached, before the program proceeds with the join point. The after advice of a particular join point runs after the program proceeds with that join point. For example, an after advice on a method-call join point runs after the method body has run, just before control is returned to the caller. Because Java programs can leave a join point ‘normally’ or by throwing an exception, there are three kinds of after advice: after returning, after throwing, and plain after (which runs after returning or throwing, like Java’s finally). Around advice on a join point runs as the join point is reached. It has explicit control over whether the program proceeds with the join point, and it may provide the join point with modified argument values.
Exposing Context in Pointcuts

Pointcuts can also expose part of the execution context at their join points. Values exposed by a pointcut can be used in the body of advice declarations. An advice declaration has a parameter list (like a method) that gives names to all the pieces of context that it uses.

The advice’s pointcut may publish the calling object, the target object, and the values of the arguments. These entities are accessed using the three primitive pointcuts this, target and args, respectively.

A named pointcut may have parameters like a piece of advice. When the named pointcut is used (by advice, or in another named pointcut), it publishes its context by name just like the this, target, and args pointcut.

Aspect

An aspect wraps up pointcuts, and advices in a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, and initializers in addition to the crosscutting members. The instantiation of aspects is controlled by AspectJ and there is no need to use Java’s new form to create new aspect instances. Since each aspect is a singleton by default, only one aspect instance is created. This means that advice may use non-static fields of the aspect, if it needs to maintain a state.

We are not aware of any other tool which utilizes aspects for the capturing process. Compared to conventional instrumentation techniques, aspects provide a clean and structured way to implement the capture functionality. One advantage of our approach is that it makes it easy to implement the tool for other aspect oriented programming languages. In addition, GenUTest automatically generates mock aspects which will be presented in Chapter 5. Mock aspects, which utilize AspectJ, assist the developer in testing units in isolation in an almost seamless manner.

1.3 JUnit

JUnit [37, 8] is one of the first frameworks that were developed for unit testing in general and Java in particular. It helped to create a standard way for writing unit tests. JUnit enables to automatically execute unit tests in a convenient manner. This section provides a brief demonstration on how JUnit tests are written. We use the classical calculator example, shown in Figure 1.3, to explain how such tests are developed.

We would like to write several unit tests to test all the methods defined in the Calculator class. A test is a regular method preceded with the @Test annotation declared in an arbitrary class. This annotation informs JUnit that the method is a test and it needs to be executed. The test method is comprised of code which executes the test and an assertion which checks whether the test result is correct.

We begin with writing a test for the add method. The test checks that the result of an add operation applied to the numbers 3 and 5 is indeed 8. The code fragment is shown in Figure 1.4.
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
    public int sub(int a, int b) {
        return a - b;
    }
    public int mul(int a, int b) {
        return a * b;
    }
    public int div(int a, int b) {
        return a / b;
    }
}

Figure 1.3: The classic calculator example.

@Test public void add3And5Equals8() {
    // code executing the test
    Calculator calc = new Calculator();
    int result = calc.add(3, 5);

    // assertion
    assertEquals(result, 8);
}

Figure 1.4: A test for the add method.

In some cases, we may want to check that under certain circumstances the code throws an exception. This is achieved by adding the expected parameter to the @Test annotation. For instance, we can add a test which checks that dividing by zero triggers an ArithmeticException as seen in Figure 1.5.

Figure 1.6 shows the feedback given to the developer after running the tests. Notice that we added another test that deliberately fails in order to show the outcome of a failing test as well.

The rest of the thesis is organized as follows. In Chapter 2 we discuss related work. Chapter 3 describes the capture phase, and Chapter 4 explains how unit tests are generated from the captured data. Chapter 5 elaborates on the creation of mock aspects. We conclude with Chapter 6 which describes our experimentation with the tool and discusses future plans.
@Test(expected=java.lang.ArithmeticException.class)
public void divideByZeroThrowsException() {
    // code executing the test
    Calculator calc = new Calculator();
    int result = calc.div(3, 0);
}

Figure 1.5: A test that checks that an exception is thrown when dividing by zero.

Figure 1.6: Unit test results reported by JUnit.
Chapter 2

Related Work

There exist various tools that automatically generate unit tests. Unit test creation in those tools
requires generating test inputs, i.e., method-call sequences, and providing test assertions which
determine whether a test passes.

There are several techniques to generate test inputs. Tools such as [3, 15, 13, 1] are catego-
rized as random execution tools. The CUT is analyzed using reflection to retrieve the methods
and their parameters. Suppose a class C has a method with a signature f(A₁, A₂,..., Aₙ), and the
method returns R. In order to test this method the tool needs to know how to construct values
of the types C, A₁, A₂,..., Aₙ. Additionally, the tool can infer that an object of type R or any
of R’s super types can be constructed, as long as it can construct C, A₁, A₂,..., Aₙ. Using these
inference rules, sequences of random method-calls can be generated automatically.

Symbolic execution tools, such as [22, 20, 2], generate a sequence of method-calls with
symbolic arguments. The tools explore the paths of each method, by analyzing the branch
conditions, and build a path condition for each path. The path conditions are solved using
constraint solvers to provide real arguments to be used instead of the symbolic ones.

Capture and replay tools, e.g. [23, 17, 14, 5], capture and record method sequences, argu-
ment values, return values, and thrown exceptions which are observed in real, or test, executions
of the software. The recorded data can be used to generate test cases and/or mock objects.

The capture and replay tools can also serve the creation of test assertions. This is done by
comparing the values obtained during the execution of the tests to the recorded values.

Test assertions can also be generated with the following approaches. Tools such as [3]
analyze exceptions thrown by the CUT during the execution of a test. They assume that a class
should not crash with an unexpected runtime exception, regardless of the parameters provided.
A test that violates this assumption is considered to uncover a bug in the CUT.

Other tools, e.g. [15, 4, 2], infer an operational model of the CUT. This operational model
consists of class invariants, and pre- and post-conditions for each method of the class. The
operational model can be constructed from either user specifications or from dynamic analysis
tools such as Daikon [27]. Daikon monitors the execution of the program under test and is able
to infer class invariants, and pre- and post- conditions. Then, the result of unit tests generated
with techniques described earlier are checked against the operational model. Violations of pre-
conditions, post-conditions, or invariants may suggest faulty behavior of the code under test.

GenUTest generates both method-calls and test assertions based on the method-call se-
quences and the values captured in real executions. Thus, it is related to capture and replay tools. In [23, 17, 14, 5] the capture process of Java programs is implemented by applying complicated instrumentation techniques on the target’s bytecode. These may include renaming interfaces, adding members and methods to existing classes, handling language specific constructs, and even modifying Java runtime libraries. The instrumentation technique used in GenUTest is quite simple and relies on weaving aspects. This is sufficient to effectively implement the capture mechanism in a natural and elegant manner. Furthermore, it is easy to implement the tool for other aspect oriented programming languages. Saff et al. [17], in their work on automatic test factoring, partition the system into two parts, a tested component T and its environment E. They limit the capturing and recording process to the interactions that occur between T and E only. These interactions are used to generate mock objects that comprise E’, the mimicked environment. During testing, T is tested with E’, and E’ can only be used for this particular partition. In addition, it is required that T will be instrumented to enable the use of mock objects. Similar techniques are used in SCARPE [14], where the replay scaffolding technique can also be used to mock the behavior of the environment.

GenUTest captures and records interactions between all objects in the system. Besides the creation of mock aspects, GenUTest also generates unit tests. Each unit test is in an independent entity, thus the developer can execute any subset of them. Moreover, their use does not require the instrumentation of the CUT.

The techniques employed in our work, in [14], and in [17], are based on method sequence object representation. In the works described in [23] and in [5], the concrete state of an invoked object before and after each method-call is captured and recorded as well using conventional instrumentation techniques. In Substra [23], the captured object states are used to infer constraints on the order of invoked method-calls. Based on these constraints new method-call sequences are generated with random values for arguments. These new method-call sequences are in fact automatically generated integration tests that exercise new program behavior. In [5] the object states and sequences are used to create entities called differential unit test cases. Their motivation and goals are similar to those discussed in our thesis. However, their differential unit test cases are not independent entities, but rather proprietary objects which are replayed using their specialized framework. Moreover, since they focus on pre and post object states, mock entities are irrelevant in their work. In addition, the use of concrete object states incurs a heavy price on the performance and storage requirements of their framework. Also, since concrete object state representation is composed of all the fields of an object, it is more sensitive to changes introduced into the unit as compared to the method sequence representation comprised of public method invocations. Thus, method sequence representation seems to be more suitable for unit testing, which perform black box testing of the unit, and for refactoring.
Chapter 3

GenUTest: The Capture Phase

In the next two chapters we describe the GenUTest tool. The current chapter describes how interactions between objects are captured and logged. We start with a brief survey of Capture And Replay (CAR) and with a short explanation of conventional instrumentation techniques.

The CAR technique enables to record software executions and to replay them later. A common use of this technique is in regression testing of graphical user interfaces. CAR tools, e.g. WinRunner [32], enable a tester to record a sequence of GUI actions in the form of a test script. The script is replayed to verify the behavior of new versions of the GUI. Those tools may also automate the comparison of actual and expected screen output.

CAR is not limited, however, to recording GUI actions performed by a user. It can also be used to record cross system interactions or inner software interactions. For instance, jRapture [18] is a Java tool which captures interactions occurring between a Java application and the system which includes GUI, files, console, and other system resources. This allows to record real executions of the software when run by beta testers. The data is then replayed, profiled, and analyzed by trained testing personnel. This process provides feedback and information regarding the application, that can not be provided by regular feedbacks and bug reports submitted by beta testers. Tools such as SCARPE [14] capture and replay inner software interactions of Java applications. Similarly to jRapture, SCARPE enables to capture events that occur in the field. These can then be used in order to generate test cases, to perform expensive dynamic analysis, and even to create unit tests.

CAR can be implemented using various approaches; we focus on implementing it using instrumentation techniques. Instrumentation is a widely used technique which changes a program in order to modify or to extend its behavior. For example, profiling tools such as VTune [34], instrument a given program with special code. When the instrumented program is executed, performance parameters are measured and recorded. The recorded data is then analyzed by the tool and assists developers in finding performance bottlenecks. Instrumentation is also used in a harmful manner by malicious software, such as viruses, and rootkits. These instrument programs and system modules with malicious code. The malicious code is executed whenever the compromised programs and system modules are executed.

We describe the instrumentation techniques used to implement CAR in the Java environment. These techniques, however, can be implemented in a similar manner (with the proper specific domain modifications) in other environments.
In order to correctly replay executions, CAR tools must be able to capture and record execution data in a comprehensive and precise manner. This requires sophisticated and extended modification of the given program’s bytecode. The modifications may include changing or replacing the Java runtime libraries, adding interfaces, adding new class members and methods to existing classes. Specific language constructs, such as reflection, callbacks, native calls, classloader, etc., must also be handled by these techniques.

3.1 Capturing in GenUTest

We implement CAR in Java by utilizing AspectJ. Incorporating aspects into a program using the weaving process is ultimately an instrumentation of the program. This is a clean and more intuitive method for instrumenting code, in contrast to other instrumentation techniques. The tradeoff, however, is that it is a weaker instrumentation mechanism than direct instrumentation techniques, which are used by other CAR tools.

To illustrate our ideas we employ a simple example which we use throughout the thesis. This is an integer stack implemented using a linked list. In addition to the conventional stack operations, the stack also supports a reverse operation, which reverses the order of the items in the stack. Figure 3.1 presents a UML sequence diagram which describes a possible scenario of the stack behavior.

In order to perform the capture phase for a given program \( P \), specific capture functionality has to be added to \( P \). The functionality is added by weaving the capture code into \( P \) at the designated join points. Since the generated unit tests are black box tests of the CUT, the advice implementing the capture code should be weaved at the following join points in \( P \): all public constructor-calls, all public method-calls, and all public read/write field-accesses. The constructor-calls, method-calls, and read/write field accesses will be considered as events for the rest of the thesis. The join points listed above can be matched using two approaches.

3.1.1 The Static Approach

In this approach we first analyze the structure of the target program in order to obtain information regarding the classes and their respective constructors, methods and fields. This can be either done by using static code analysis of the program’s code or by using reflection. With the signatures of all constructors, all methods and all public read/write field-accesses, we can easily define pointcuts that match each and every one of those events.

3.1.2 The Dynamic Approach

The dynamic approach does not need an extra step in order to pre process the target program to obtain information about the constructors, methods and fields. Pointcuts are defined a general manner that enable them to match all the required events. However, the tradeoff of utilizing such

---

1The numbers in italic are used to denote event intervals which are introduced in Section 3.1.2. Note that in order to make it easier for the reader to follow the example, we use dashed lines to denote return from a call even for the constructor.
Figure 3.1: A sequence diagram describing a scenario of the stack behavior.

general declarations is apparent in the repeated use of reflection whenever an event is captured in order to discover the arguments of the event.

In GenUTest we chose the dynamic approach and we define several pointcut definitions in order to match those events. For instance, to match all the public constructor-calls, we define the following pointcut:

\[
\text{pointcut publicConstructorCall(): call(public \ldots .new(\ldots)) \&\& \neg \text{within(GenUTest.\ldots)}};
\]

The following pointcut declaration matches all public method-calls:

\[
\text{pointcut publicMethodCall(): call(public \star \star (\ldots)) \&\& \neg \text{within(GenUTest.\star)}};
\]

In order to match all public write field-accesses we define:

\[
\text{pointcut publicSetter() : set(public \star \star \star (\ldots)) \&\& \neg \text{within(GenUTest.\star)}};
\]

Finally, all of the public read field-accesses are matched using the declaration:

\[
\text{pointcut publicGetter() : get(public \star \star \star (\ldots)) \&\& \neg \text{within(GenUTest.\star)}};
\]
The \texttt{within\{GenUTest.\}} join point ensures that only designated join points within $P$ are matched. The two wildcards (‘*’ and ‘..’) enable matching a set of signatures.

The pointcuts are accompanied by advices which capture and record the events. Moreover, the advices need to record the execution order of the events. This can be achieved by simply time stamping the events. However, this is not enough to fully describe the execution relationships of two different events. Let us examine, for instance, the events \texttt{push(2), addFirst(2),} and \texttt{push(3)} shown in Figure 3.1. A time stamp assigned to the events just before they are executed will provide the time stamp sequence: $t$, $t+1$, $t+2$. Assigning a time stamp to the events right after their execution ends will provide the time stamp sequence: $t+1$, $t$, $t+1$. Neither of the time stamp sequences correctly portray the execution of the first two events. This is due to the fact that the second event \texttt{addFirst(2)} occurred \textbf{during} the execution of the first event \texttt{push(2)}.

Time is represented by a global sequence number which is incremented \textit{before} an event begins execution and \textit{after} it finishes execution. The interval \textit{[before, after]} is called the \textit{event interval}. Using the \textit{before} value of the event interval, an order relation can be applied to events, while the containment relation of two intervals expresses when event $b$ occurs during event $a$.

The advices are implemented using the around advice mechanism which enables GenUTest to record the time before an event, execute it, record the event and record the time after it ends. Capturing an event involves recording the event’s signature and the target object of the call. Returned values and thrown exceptions are recorded as well. The type of the arguments and the return value can be one of the following: primitive, object, or array. For instance, the attributes of the \texttt{IntStack_2.push(3)} (cf. Figure 3.1) are: the AspectJ signature, which consists of full name (\texttt{IntStack_2.push()}), arguments type (int), access modifier (public) [38], and the return type (void); the target object (IntStack_2); the arguments values (3); the return value (none); and the exception thrown by the method (none).

The instrumented program $P'$ is executed and the actual capturing begins. The capture code, which is specified by the advice, is responsible for obtaining the above mentioned attributes. This is achieved using the AspectJ reflective construct (\texttt{thisJoinPoint}). This construct provides access to the following methods:

1. \texttt{getArgs()} - The method returns an array of objects (\texttt{Object[]}) which can be traversed in order to explore the values of the arguments passed to methods and constructors.

2. \texttt{getSignature()} - This returns a reference to an instance of one of the Signature Class types: FieldSignature, MethodSignature, ConstructorSignature. The signature objects allow GenUTest to extract all the signature information as described in the previous section.

3. \texttt{getThis()} - For advices used at call join points, as GenUTest does, the method \texttt{getThis()} returns a reference to the instance invoking the event. For example, let us examine the method-call \texttt{addFirst(2)} invoked during the execution of the method \texttt{push(2)} as shown in Figure 3.1. The advice is executed at the call location. Thus, \texttt{getThis()} will return a reference to the object \texttt{IntStack_2}. In case an event is invoked within a static method, \texttt{getThis()} returns null.

4. \texttt{getTarget()} - Similarly to \texttt{getThis()}, this method returns a reference to the target object of the event. In the previous example, it will return a reference to the object \texttt{lst}. 

14
In case a static method is invoked or a static field is accessed, \( \text{getTarget()} \) returns null.

### 3.2 Object Identification

Objects may play various roles in an event. They can be the target object of the event, an argument, or a return value. In GenUTest, objects are represented by an entity called an object record. This entity enables GenUTest to keep track of the objects encountered during capture and to distinguish between them. An object record is comprised of three fields: object ID, static type ID, and dynamic type ID.

#### 3.2.1 Object ID

An object ID is a positive integer that uniquely identifies an object. The object ID is calculated in the following manner. GenUTest initializes a global counter before the capturing begins. Every time GenUTest encounters an object, it retrieves its object record from an object repository. The object repository is an instance of an IdentityHashMap [33], which maps objects to their corresponding object record. Unlike regular maps, where two keys \( k1 \) and \( k2 \) are considered equal if and only if \((k1 == \text{null} ? k2 == \text{null} : k1.equals(k2))\), in the IdentityHashMap two keys are considered equal if and only if \( k1 == k2 \). In case the object is encountered for the first time, GenUTest increments the global counter, creates an object record, assigns it to the object, and adds the pair to the repository.

#### 3.2.2 Static & Dynamic Type IDs

The dynamic type id is obtained by querying the type repository which maps class objects to IDs. The type repository obtains the class of the object using the \( \text{getClass()} \) method and returns its ID when it exists. Otherwise, it assigns a unique ID to the class object. However, when the object is null, its type can only be determined by examining the signature of the event. A null object may be encountered in the following cases:

1. A static method is invoked.
2. A static field is accessed.
3. A null value is passed as an argument to a method.
4. A null value is returned from a method.
5. A null element in an array is accessed.
3.3 Event Log

After the attributes of the events are obtained, they are serialized and logged. In addition to specific event data, the serialized events contain references – represented as ids – to signature records, and type records. These will be discussed shortly.

The serialization is achieved using the XStream serialization library [42]. The library supports the serialization of complex nested objects which is especially important for arrays.

Figure 3.2 is an XML fragment of a serialized method-call. The returnValue element records the return value of the method along with the ID of the type record which is stored in the type repository. The return value can represent primitive types, objects, and arrays. In the example, the type of the return value is an integer and its value is 2. The eventInterval element is the event interval. The signature ID points to the signature of the event which is stored in the signature repository. Finally, the objectRecord element holds the instanceID referring to the target object of the event (IntStack in the example), and the ids of its static and dynamic types.

```xml
<MethodCall>
  <returnValue class="PrimitiveTypeValue">
    <value>2</value>
    <typeID>6</typeID>
  </returnValue>
  <returnVariableName>Integer_11</returnVariableName>
  <eventInterval>
    <start>37</start>
    <end>42</end>
  </eventInterval>
  <signatureID>12</signatureID>
  <objectRecord>
    <instanceID>2</instanceID>
    <staticClassID>13</staticClassID>
    <dynamicClassID>13</dynamicClassID>
  </objectRecord>
</MethodCall>
```

Figure 3.2: An XML representation of a Method-Call event.

3.3.1 Signature Record

A signature record is held in the signature repository and describes the event’s signature. GenUTest has three types of signature records: ConstructorSignatureRecord, MethodSignatureRecord, and GetSetSignatureRecord. All signature records contain the following fields:

1. **Type ID** - The id of the static type of the target object.
2. **Signature String** - A string representation of the signature.
3. **Modifier** - The Java access modifiers of the constructor, method, or field. These can encode various properties such as transient, public, static, volatile, synchronized, etc.

ConstructorSignatureRecord and MethodSignatureRecord describe the AspectJ signature of a constructor-call and method-call events, respectively. Both records contain information describing the type of the events’ arguments. The MethodSignatureRecord also contains the type of the event’s return value. The GetSetSignatureRecord, which describes the AspectJ signature of a field read/write access event, contains the name and the type of the field.

Following is an XML fragment of the signature of the method-call `IntStack.pop()`

```
<entry>
    <long>12</long>
    <MethodSignatureParser>
        <methodName>pop</methodName>
        <returnTypeID>6</returnTypeID>
        <signature>int IntStack.pop()</signature>
        <classID>13</classID>
        <accessDescription>
            <modifiers>1</modifiers>
        </accessDescription>
    </MethodSignatureParser>
</entry>
```

### 3.3.2 Type Record

A type record is stored in the *type repository* and describes the type using the following fields:

1. **Type ID** - The unique id associated with the type.
2. **Type Name** - The Java name of the type.
3. **Package Name** - The package name in which the type resides, or null if the name is not available.
4. **Primitive Indicator** - Specifies whether the type is a primitive type.
5. **Array Indicator** - Specifies whether the type is an array type.
6. **Component Type ID** - Specifies the type of an array components.
7. **Comparison Indicator** - Indicates whether an instance of the type can be compared using the class comparison method `equals`.
8. **Modifier** - The Java access modifiers of the type.

The following XML fragment is a type record entry which describes the `IntStack` type.
3.4 Special Cases

3.4.1 Object Dependency

Most objects are created via an explicit call to a constructor. In some cases, object creation is implicit rather than explicit. String objects, for example, can also be initialized with assignment, i.e., no constructor is involved. GenUTest ensures that these objects are explicitly initialized via a constructor call. This matter is discussed in length in Section 3.4.2. Iterator objects can only be initialized by calling the iterator() method of a collection. GenUTest distinguishes between objects whose construction is explicit and those whose construction is implicit. Objects which are explicitly constructed are named self dependent objects, whereas the other objects are called event dependent objects. This dependency trait is completely transparent for other events invoked on the object.

The construction event of self dependent objects is accessible to AspectJ and hence to GenUTest. Event dependent objects are introduced as a result of a method-call. GenUTest examines the return value of each method-call and adds it to the event dependent repository in case it is an event dependent object. Event dependent objects which are arrays require additional handling as described in the following section.

Element Array Dependency

Let us assume that objArr is an event dependent array that contains elements that are of any non-primitive type. While it is clear to the developer that objArr[2] is an element of the array objArr, it is not apparent to GenUTest. For GenUTest, those are simply two different references; one that references an object of type Object[] and the other is to an object of type String. GenUTest can not deduce that those two references are related. The relation needs therefore to be formulated in some manner, and this is achieved by introducing an assignment event. This event expresses the assignment of an object of a non primitive type to an object referencing it. It supports the following assignment statements:

1. someReference = someObject
2. someReference = someObjectArray[someIndex]
When an array of non-primitive types is first encountered, GenUTest pairs each element of the array with an assignment event. Each pair is then added to the object event dependency map. If an element is an array then the process is applied in a recursive manner.

### 3.4.2 Strings

The Java String type is a non-primitive one representing character strings. However, Java programmers can use instances of this class in a similar manner to the way primitive types are used. For instance, String objects can be initialized with the assignment operator (=):

```java
String str = "abc";
```

which is equivalent to:

```java
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

The equivalence is due to the fact that “Strings are constant; their values cannot be changed after they are created and so they can be shared” [40]. When assignment is used a String constructor is implicitly called.

This implicit constructor-call can not be matched by an AspectJ join point. The ramification of this fact is that during the capture phase, GenUTest may encounter String objects whose construction had not been recorded. Thus, the state of those objects can not be restored properly in the generation phase.

This issue is addressed in the following manner. GenUTest checks whether any arguments of the event or the target object of the event are String objects whose constructors event were not recorded. The checks are performed by querying the object repository to see if the String object already exists in it. If it does, then either it was constructed properly or GenUTest had already resolved the problem when it was first encountered. For every String object that does not exist in the repository, GenUTest generates a synthetic constructor event, which simulates the creation of a String object, and accordingly increments the sequence number. This event is then passed on to the event logger, which in turn records it as if it were a regular constructor event. The event logger can not distinguish between a real String constructor event and a synthetic one.

In case one of the arguments is an array then GenUTest applies the above process in a recursive manner.

### 3.4.3 Class Object

A class object can be obtained in various ways. The most common one is calling an object’s `getClass` method:

```java
String a = "a";
Class c = a.getClass();
```

Class objects can also be obtained using the Reflection API [36]:

19
Class c = null;
try {
    c = Class.forName("String");
} catch (ClassNotFoundException ex) {
    // handle exception case
}

The simplest way to obtain a class object is by accessing the object’s `class` field:

Class c = String.class;

In the first two approaches the reference ‘c’ to the String class is an event dependent object, i.e., it is obtained via a method-call. The third approach, however, obtains the reference ‘c’ to the String class via accessing the static field `class`. This static field is not initialized in the course of the program, but rather by the JVM when the class is loaded. This initialization event can not be accessed by AspectJ, and therefore can not be accessed by GenUTest as well. Thus, GenUTest can not capture the initialization of the `class` field and when the reference ‘c’ is later encountered, GenUTest can not determine how it was initialized. To resolve this problem, GenUTest monitors the creation of objects of type ‘class’. When such an object is encountered, GenUTest extracts the class name and sets the reference name to the represented object accordingly.

The capturing process ends after all events have been logged. The log is used in the generation phase to create unit tests.
Chapter 4

GenUTest: The Unit Test Generation Phase

In this chapter we explain how unit tests are generated from the captured method-calls. Unit tests are created only for those methods that can be examined, i.e., methods that either return a value or throw an exception. In the example, when IntStack serves as the CUT, GenUTest generates a unit test only for the pop() method-call (cf. Figure 4.1).

```java
1 @Test public void testpop1() {
2     // test execution statements
3     IntStack IntStack_2 = new IntStack(); // mi [1,4]
4     IntStack_2.push(2); // mi [5,8]
5     IntStack_2.push(3); // mi [9,12]
6     IntStack_2.reverse(); // mi [13,24]
7     int intRetVal6 = IntStack_2.pop(); // mi [25,28]
8
9     // test assertion statements
10    assertEquals(intRetVal6,2);
11 }
```

Figure 4.1: Unit test generated for pop() method-call.

Test generation is a two step operation followed by a post-processing stage. In the first step GenUTest generates the Java statements that execute the test. In the second step GenUTest generates assertion statements that determine whether the test has passed.

4.1 Step I: Test generation

The Java statements generation algorithm contains two mutually recursive procedures, `restoreObjectState` and `generateStatement`. The procedure `restoreObjectState` selects the method-calls which are needed to execute the test, whereas `generateStatement` generates Java statements
that execute those method-calls. In the stack example, the method-calls `new IntStack()`, `push(2)`, `push(3)`, `reverse()`, and `pop()` are selected by `restoreObjectState`, and their corresponding test statements (lines 4 to 8 in Figure 4.1) are generated by `generateStatement`.

### 4.1.1 The `restoreObjectState` procedure

This procedure obtains as input an object id (which represents the object `obj`) and a time stamp `T`. The **concrete state** of the object at a given point of time `T` is defined by the values of its state variables at that time. We represent an object state using *method-sequences* [21]. In GenUTest this representation consists of the sequence of captured events that had been invoked on the object till `T`. More formally, we define:

**Definition 1**  
\[ \text{State}_T(obj) = \langle \text{creation\_event}_{t_1}, \text{event}_{t_2}, ..., \text{event}_{t_n} \rangle, \]  
where \( \text{creation\_event}_{t_1}, \text{event}_{t_2}, ..., \text{event}_{t_n} \) are all the events that had occurred on object `obj` at times \( t_1, t_2, ..., t_n \), respectively, and satisfy \( t_1 < t_2 < ... < t_n < T \).

In order to test `obj` at its correct state, all the events invoked on `obj` prior to `T` need to be reinvoked. Suppose that the method `m()` had been invoked at time `T`. Using the event intervals, `restoreObjectState` reconstructs all method-calls that had been invoked prior to that specific invocation. For example, let us refer to the object `IntStack_2` in Figure 3.1. To invoke the `reverse()` method which occurred at time stamp 13 on the object `IntStack_2`, the methods `new IntStack()`, `push(2)`, and `push(3)`, which had occurred before time stamp 13, must be reinvoked.

Notice that the creation event is either a constructor event or a method event applied to another object and is determined by the dependency trait of the object. In case the object is a dependent object, the procedure must be applied to the object it depends on as well.

### 4.1.2 The `generateStatement` procedure

This procedure generates a Java statement according to the event type.  
For a method-call event the procedure generates a statement of the following form:

\[ \text{<return variable>} = (\text{<return type>})<\text{object reference}>. \]  
\[ \hspace{1cm} \text{<method name>}(<\text{arg #1}, ..., \text{arg #n}>); \]

The **object reference** is a string representation of the target object. It is formed by concatenating the object type (obtained from the method signature) with the object id. For example, an `IntStack` object with the object id 2 is represented as “IntStack_2”. If the method-call is a static one, then the object reference is the name of the class. The **return variable** name is formed by creating a unique string representation. The arguments’ array is traversed by the procedure to obtain the values of the arguments. The representation depends on the argument’s type:

1. A primitive value is represented according to Java rules. For instance, float values are suffixed with the character ‘f’. Character strings are handled according to Java String rules (e.g., newlines and tabs are replaced with ‘\n’ and ‘\t’, respectively) and then are surrounded by a pair of quotes.

22
2. An object is represented by an object reference. To ensure that the object is in the correct state when it is used, `restoreObjectState` must be invoked with the relevant arguments, which in turn leads to the invocation of `generateStatement`.

3. An array is represented by the following code fragment:
   ```java
   new <ArrayType> { <elem #1>, ..., <elem #n> },
   ```
   where elements are represented according to their type.

For a constructor-call event the procedure generates a statement of the following form:

   ```java
   <object reference> = new <classname>(<arg #1,..., arg #n>);
   ```

   The arguments are processed in the same manner arguments are processed when generating a method-call Java statement.

For a read field access event the procedure generates a statement of the following form:

   ```java
   <variable> = <object reference>.<field name>
   ```

   If the field is a static field, then the object reference is the name of the class. This also applies to write field access events.

For a write field access event the procedure generates a statement of the following form:

   ```java
   <object reference>.<field name> = <arg #1>
   ```

   The argument `arg #1` is a single argument in an argument list and is handled similarly.

For an assignment event the procedure generates a statement of the following form:

   ```java
   <object reference1> = <array reference>[<array index>]
   ```

In the following example we demonstrate how both procedures work on a more complicated example involving objects. Figure 4.2 presents the method-calls occurring at consecutive event intervals for three different objects: obj1, obj2, and obj3.

Suppose GenUTest encounters the method-call `obj1.foo1(obj2)` which had occurred at time stamp 31. In order to invoke the method-call, GenUTest must restore `obj1`, the target object, to its correct state at time stamp 31. This is achieved by the procedure `restoreObjectState` which selects the constructor-call `obj1 = new Type1()` that had occurred at time stamp 1. Then, the procedure `generateStatement` is invoked in order to generate the Java statement for the method-call `obj1.foo1(obj2)`. During the execution of the generation procedure, it encounters the argument `obj2` and in order to restore this object to its correct state at time stamp 31, the procedure invokes `restoreObjectState`. Then, `restoreObjectState` selects the constructor-call `obj2 = new Type2()` and the method-call `obj2.goo1(obj3)` which had occurred at time stamps 5 and 21, respectively. For the latter method-call, `generateStatement` is invoked in order to generate its corresponding Java statements. It encounters the argument `obj3`, and invokes `restoreObjectState` in order to restore the argument to its correct state at time stamp 21. Eventually, the algorithm generates the statements as shown in Figure 4.3.

After generating the statements, the algorithm performs some post processing tasks. One of those tasks is the removal of redundant statements. For example, when replacing the method-call `obj1.foo1(obj2)` in the previous example with the call `obj1.foo1(obj2, obj3)`, then the statements at lines 2 and 4 in Figure 4.3 would be generated twice. This leads to an incorrect sequence of statements which in some cases might affect the state of the objects. The post processing task detects and disposes of such redundant statements.
Figure 4.2: Method-calls invoked on the objects obj1, obj2, and obj3.

1 Type1 obj1 = new Type1();
2 Type3 obj3 = new Type3();
3 Type2 obj2 = new Type2();
4 obj3.initialize();
5 obj2.goo1(obj3);
6 obj1.foo1(obj2);

Figure 4.3: Statements generated to restore the correct state of obj1 and obj2.

4.2 Step II: Test Assertion

The assertion statements generated by GenUTest determine whether the test has passed. There are two cases to handle: 1) the method returns a value and, 2) the method throws an exception.

4.2.1 Case I: A Value is Returned

GenUTest generates in this case statements to compare the value returned by the test ($value_{test}$) with the captured return value ($value_{captured}$). The structure of the statements depends on the values' types.

Primitive values can be directly compared using one of JUnit's asserts statements. In the example, intRetVal6 ($value_{test}$) is compared to 2 ($value_{captured}$) (cf. Figure. 4.1, line 11).

When the returned values are objects or arrays, the comparison is more complicated. In the case of objects, GenUTest generates statements that restore the state of $value_{captured}$ (as described in Section 4.1). Then, GenUTest queries the type repository to check whether an equals method had been implemented for the objects being compared. If equals exists, GenUTest generates a JUnit $assertTrue$ statement which simply checks equality by invoking the equals method. Otherwise, special statements are generated to compare the concrete
state of the two objects. This is achieved using the Java reflection mechanism, which enables to discern information about the objects’ fields. The discovered fields are compared according to their types, using the `CompareToBuilder` class of the `Commons Lang` library [26].

Since the `CompareToBuilder` class does not support array comparison, a slightly different approach is taken. If the arrays contain primitive types or Strings, the comparison is performed by the `Arrays.equals` method [24]. Otherwise, GenUTest provides a special method. This method accepts as input two objects \((a \text{ and } b)\) to be compared. It first ensures that the two objects are arrays of the same length. It then recursively traverses both arrays, performing a deep comparison.

### 4.2.2 Case II: Thrown Exception

For the second case, when a method throws an exception, GenUTest generates a statement that informs JUnit that an exception of a specific kind is to be thrown. This is achieved by adding the `expected` parameter to the `@Test` annotation as follows: `@Test(expected= ExceptionClassName.class)`. The exception kind is obtained from the captured attributes of the method-call. For example, suppose the method `pop()` is invoked on a newly created object `IntStack_3`. This is an attempt to remove an item from an empty stack. Thus, an exception of type `NoSuchElementException` is thrown. GenUTest informs JUnit to expect an exception of this type. Figure 4.4 presents the generated code for this scenario.

```java
1  @Test(expected= NoSuchElementException.class)
2  public void testpop2() {
3      // test execution statements
4      IntStack IntStack_3 = new IntStack();
5      IntStack_3.pop();
6  }
```

Figure 4.4: Unit test generated for exception throwing `pop()` method-call.
Chapter 5

Generating Mock Aspects

This chapter describes what mock objects are. We explain the pattern virtual mocks and mention the benefits of mock object frameworks. We then introduce a new concept, namely mock aspect, explain its advantages, and describe how it is generated.

5.1 Mock Objects

Testing a unit in isolation is one of the most important principles of unit testing. Most units, however, are not isolated, and the dependencies between units complicate the test process, and sometimes the developer’s intervention is required. A common approach to deal with this issue is the use of mock objects [12].

A mock object is a simulated object which mimics the behavior of a real object in a controlled way. It is specifically customized to the Class Under Test (CUT). It can be programmed to expect a specific sequence of method-calls with specific arguments, and to return a specific value for each method-call. A mock object has the same interface as the real object it mimics. As a rule of thumb, a mock object replaces a real object, when the latter satisfies any of the following properties:

1. It supplies “sensed” results (i.e., results that depend on an external environment, e.g., the current time or the current temperature).
2. It has states that are difficult to create or reproduce (e.g., a network error).
3. It is slow (e.g., a complete database, which would have to be initialized before the test).
4. It does not yet exist or may change behavior.
5. For testing purposes, it will contain redundant information or methods.

The references of a CUT to real objects can be replaced with references to mock objects, leaving the CUT unaware of which objects it addresses. In order to support mock objects creation, the CUT’s code must be modified as follows. Code fragments which explicitly create instances of a real object must be modified in order to enable the creation of mock object instances as well. The modified code will create real objects when normal program behavior is
desired, and mock objects when testing the CUT in isolation is desired. A common method to achieve this is to employ a Abstract Factory Pattern [7].

### 5.1.1 Abstract Factory Pattern

Objects that may be mocked are created using an Abstract Factory [7]. The factory returns a reference to either a real object or to a mock object, according to the desired usage. One way to implement this is to instantiate a reference to a factory interface with either a RealObjectFactory instance or a MockObjectFactory instance. Figure 5.1 shows a code snippet that creates an object using the pattern.

```java
// regular object instantiation
AbstractList t = new LinkedList();
t.addStart(2);

// instantiation with Abstract Factory
// ...
// Set factory in program’s initialization phase
ObjectFactory objFactory = new MockObjectFactory();
// ...
AbstractList t = objFactory.createList();
t.addStart(2);
```

Figure 5.1: Object creation using the factory pattern.

### 5.1.2 Virtual Mocks

While the factory pattern assists in modifying the code of the CUT in an elegant and minimal manner, the code still needs to be changed. Fortunately, there exists another pattern that harnesses the virtues of AOP. The Virtual Mock Pattern [41] utilizes aspects to enable the usage of both real and mock objects in a seamless manner, without having to modify the code of the CUT. This is achieved by defining pointcuts that match real object method-calls. The associated advices which are performed at those pointcuts redirect the method-calls to the mock objects. The CUT is completely unaware of this intervention and is “fooled” into calling the mock object instead. Thus, virtual mocks enable us to test units in isolation using mock objects without having to modify any client code at all.

### 5.2 Mock Object Frameworks

Developing mock objects is a hard and tedious process. This involves activities such as declaring new classes, implementing methods, and adding lots of bookkeeping code. The process can be simplified using mock object frameworks. Such frameworks (e.g., EasyMock [28]) do the job for us. This is achieved as follows:
1. A proxy object is created, with the same interface as the object being mocked. In EasyMock, it is created using the `createMock` method which obtains as input an interface object.

2. The proxy object is provided with a sequence of method-calls and with the return values. In EasyMock, this is performed using a sequence of statements in the form of `expect(<method-call>).andReturn(<return value>).` In those statements, `<method-call>` denotes the expected method-call and `<return value>` denotes the expected return value of the call.

3. The proxy object is put in replay mode (using the `replay` statement) and is ready to interact with the CUT.

4. Finally, the CUT is initialized and provided with a reference to the proxy object, which is called by the CUT throughout the test. Note, that we may need to modify the CUT’s code in order to support its interaction with the proxy object.

Figure 5.2 demonstrates how the List object is mocked using EasyMock in the test `testpop1`. The use of EasyMock required modifying the code of `IntStack` in order to support `IntStack`’s interaction with the proxy object.

Manually implementing the above mock behavior requires declaring a new class, implementing the methods, and adding a lot of bookkeeping code. Clearly, the framework alternative is elegant and more maintainable.

### 5.3 Mock Aspects

We make use of the advantages of both mock objects and virtual mocks. We have defined a new concept, the mock aspect. A mock aspect is an aspect which intervenes in the execution of the CUT. Being an aspect, it has pointcuts and advices. The pointcuts match method-calls invoked by the CUT on real objects. The advices directly mimic the behavior of the real object, as opposed to virtual mocks, which act as mediators to the mock objects. GenUTest automatically generates mock aspects. Once created, the mock aspects can easily be integrated with the CUT to enable testing it in isolation.

Figure 5.3 illustrates two kinds of method-calls. An invocation of a CUT method from within the unit test is called an incoming method-call. An outgoing method-call is an invocation of a method in some other class from within the CUT. In the example (cf. Figure 3.1), the incoming method-calls of `IntStack` are: `new IntStack()`, `push(2)`, `push(3)`, `reverse()`, and `pop()`, whereas its outgoing method-calls are: `addFirst(2)`, `addFirst(3)`, `get(0)`, `get(1)`, etc.

The mock aspect has pointcuts which match outgoing method-calls. For each method in `object`, `1 ≤ i ≤ n`, there exists a different pointcut. Each pointcut matches all the outgoing method-calls to a specific method. For example, all outgoing method-calls to the method `addFirst()` are matched by a single pointcut declaration, and are handled by a single advice. This advice mimics the effect of all these outgoing method-calls. Thus, it needs to know
public void testpop1WithEasyMock() {
    // step #1
    MyAbstractLinkedList mock =
    createMock(MyAbstractLinkedList.class);

    // step #2
    mock.addFirst(2);
    mock.addFirst(3);
    expect(mock.createNewInstance()).andReturn(mock);
    int[] arr = new int[] { 3, 2 };
    for (int i = 0; i < 2; i++) {
        expect(mock.size()).andReturn(2);
        expect(mock.get(i)).andReturn(arr[i]);
        mock.addFirst(arr[i]);
    }
    expect(mock.size()).andReturn(2);
    expect(mock.removeFirst()).andReturn(2);

    // step #3
    replay(mock);

    // step #4
    IntStack IntStack_2 = new IntStack(mock);
    // test code
    IntStack_2.push(2);
    IntStack_2.push(3);
    IntStack_2.reverse();

    // test assertion
    int intRetVal6 = IntStack_2.pop();
    assertEquals(intRetVal6, 2);
}

Figure 5.2: Testing pop with the use of the EasyMock framework.

which particular outgoing method-call to mimic. Also, the pointcut needs to enforce that only outgoing method-calls are intercepted.

Before continuing, we provide some definitions and observations.

**Definition 2** \( mi(A()) \) is the event interval of method-call \( A() \), i.e., \( [before_A, after_A] \).

**Definition 3** Method-call \( A() \) contains method-call \( B() \) if \( mi(A()) \) contains \( mi(B()) \), i.e., \( [before_A, after_A] \supset [before_B, after_B] \).

Following these definitions, we observe that:

2. An outgoing method-call of the CUT is contained in **exactly one** incoming method-call. An incoming method-call, on the other hand, may contain several outgoing method-calls.

For example, the outgoing method-calls `get(0)` and `get(1)` are contained in the one incoming method-call `reverse()`, while `reverse()` contains several other outgoing method-calls, besides those two.

**Definition 4** $\text{Outgoing}(I())$ is the ordered sequence $< I_{o_1}(I()), I_{o_2}(I()),..., I_{o_n}(I()) >$, where $I()$ is an incoming method-call and $I_{o_1}(I()), I_{o_2}(I()),..., I_{o_n}(I())$ are all the chronologically ordered outgoing method-calls contained in $I()$.

Suppose the outgoing method-call $o()$ is contained in the incoming method-call $I()$, and suppose that it is the $j^{th}$ element in $\text{outgoing}(I())$. Then, $o()$ is uniquely identified by the pair $(\text{mi}(I)), j)$. 

In Figure 3.1 there are four outgoing method-calls to the method `addFirst()`. The first outgoing method-call, `addFirst(2)`, is contained in the incoming method-call `push(2)`. Hence, $\text{mi}(\text{push}(2))$ is [5, 8], $\text{outgoing}(\text{push}(2))$ is $< \text{addFirst}(2) >$, and `addFirst(2)` is uniquely identified by the pair ([5, 8], 1). Similarly, the second outgoing method-call `addFirst(3)` is uniquely identified by the pair ([9, 12], 1). Both the third and fourth outgoing method-calls, `addFirst(3)` and `addFirst(2)`, are contained in the incoming method-call `reverse()`. Thus, $\text{mi}(\text{reverse})$ is [13, 24] and $\text{outgoing}(\text{reverse}())$ is $< \text{new, get(0), addFirst(3), get(1), addFirst(2)} >$. The outgoing method-calls, `addFirst(3)` and `addFirst(2)`, are uniquely identified by the pairs ([13, 24], 1) and ([13, 24], 2), respectively.

In order to identify a specific outgoing method-call, the advice needs to:

1. Know all the incoming method-calls of the CUT.

2. Keep track of the outgoing method-calls sequence for each incoming method-call.

The mock aspect generation algorithm works as follows:
@Test public void testpop1() {
    IntStackMockAspect.setMI(1, 6);
    IntStack IntStack_2 = new IntStack(); // #1 [1,6]
    IntStackMockAspect.setMI(7, 14);
    IntStack_2.push(2); // #2 [7,14]
    IntStackMockAspect.setMI(15, 22);
    IntStack_2.push(3); // #3 [15,22]
    IntStackMockAspect.setMI(23, 46);
    IntStack_2.reverse(); // #4 [23,46]
    IntStackMockAspect.setMI(49, 62);
    int intRetVal6 = IntStack_2.pop(); // #5 [49,62]
    assertEquals(intRetVal6,2);
}

Figure 5.4: Unit test for pop() method-call with helper statements.

1. For each incoming method-call \( I() \) of the CUT, \textit{outgoing}(I()) is calculated.

2. Each outgoing method-call is uniquely identified.

3. The mock aspect code is created.
   This requires generating the following: aspect headers, pointcuts, bookkeeping statements, and statements that mimic the outgoing method-call. The bookkeeping statements are responsible for uniquely identifying the outgoing method-calls. These statements include matching event intervals of the incoming method-calls and maintaining inner counters to keep track of the sequence of outgoing method-calls. For an outgoing method-call that returns a primitive value, the statement mimicking its behavior is one that returns the value. When an object is returned, it needs to be in the correct state. This is achieved by using the procedure \textit{restoreObjectState} described in Section 4.1.

4. A helper statement that updates the mock aspect with the event interval of the current incoming method-call prior to its invocation is added in the unit test as shown in Figure 5.4.

Figure 5.5 shows a code snippet of the mock aspect for the CUT \texttt{IntStack}. This code mimics outgoing method-calls to the method \texttt{get()}. 
// ensure that only CUT outgoing methods are intercepted
// and prevent advices from intercepting themselves
pointcut restriction(): !adviceexecution() &&
this(IntStack) && !target(IntStack);

int around(): restriction() &&
call(Object java.util.LinkedList.get(int)) {
// match current incoming event interval [before,after]
// to associated incoming interval [13,24]
if (before == 13 && after == 24) {
  // match inner counter
  if (innerCounter == 1) {
    innerCounter++;
    return 3;
  }
  // match inner counter
  if (innerCounter == 2) {
    innerCounter++;
    return 2;
  }
}
Figure 5.5: A code snippet of the mock aspect generated for IntStack.
Chapter 6

Conclusion

In this concluding chapter, we describe some experimentations we have done with the tool and the tool’s limitations and capabilities. Finally, we describe some directions for future work.

6.1 Experimentation

During the development of GenUTest, we applied it on some small programs. Later, as the tool supported more features and became more robust, larger examples were used.

For each experiment we first have to instrument the program $P$ with GenUTest’s capture code. The instrumented program $P'$ is then executed and the events are captured and logged. For each program $P$ we name its run $R$. When $P'$ ends, GenUTest processes the recorded events and generates unit tests and mock aspects for each class. These are then executed using JUnit. The generated unit tests are evaluated with respect to the specific run $R$ from which they have been derived. The evaluation compares the code coverage obtained by $R$ and by the unit tests. The coverage is measured using the EclEmma [29] Eclipse plugin. Ideally, both code coverage measures will be similar.

Following we describe some of our experimentations.

We applied GenUTest on the IntStack example described in 3.1. A module test was designed to examine the IntStack class in a scenario based on the one described by the sequence diagram in Figure 3.1. GenUTest generated two unit tests for the IntStack class. In this simple example the unit tests obtained the same coverage as the module test.

Then we applied GenUTest on a small open source project called NanoXML [39]. It is an XML parser consisting of 24 classes and about 7,700 lines of code. We wrote a small ‘system’ test that parses and prints a small XML file. When executed normally, the test obtains a code coverage of 19.4%. When instrumented with GenUTest 324 unit tests were generated. Execution of the unit tests achieves code coverage of 14.6%. The gap of 4.8% is mainly due to the fact that events which are not testable according to GenUTest may not be invoked in the unit tests. When the testability criterion in GenUTest is loosened, e.g., when every event is tested, GenUTest generates 435 unit tests. These unit tests achieve a coverage of 17.6%. The remaining 2% gap can be explained by partially handling of inner classes.

Another project we experimented with is a single threaded ATM simulation [23]. The
project consists of approximately 40 classes and about 2100 lines of code. Execution of the system test provided with the project obtained a code coverage of 14.3%. GenUTest generated 26 unit tests, achieving a code coverage of 14.2%. In this experiment, GenUTest was able to generate test cases for almost all the scenarios executed during the system test. We hope to be able to repeat such results in other projects as well.

Finally, we applied GenUTest to a medium sized project called JODE (Java and Optimizing Development Environment). JODE is an open source project [35] which spans approximately 35 thousand lines of code. We used JODE to examine how GenUTest handles large programs with a huge amount of events. GenUTest was not designed to address scalability and performance issues apparent in such a project. The tool requires further development in order to be able to cope with projects on a larger scale. This is out of the scope of this thesis, and is described in Section 6.3. Nevertheless, we used GenUTest with JODE, which was not a trivial task. Manual intervention was required in order to successfully instrument JODE with GenUTest’s capture functionality. The instrumentation of some methods increased their size in a significant manner which surpassed the 64k limit Java imposes on the size of methods. Another problem with the instrumentation process was the increase of code size in condition blocks. In some cases, the size of the blocks was too big for a branch statement. In both cases, methods and conditional blocks were divided into other methods and conditional blocks, respectively. Other issues concerning the instrumentation of class hierarchy with static arrays, objects, etc. could not be handled gracefully by the AspectJ compiler. However, after some tinkering with the code, those issues were resolved as well.

We used the instrumented version of JODE to decompile the bytecode of the calculator class which is shown in Figure 1.3. The execution consumed a lot of memory and took a lot of time. It also uncovered some bugs and errors in GenUTest. These were fixed in a quick and dirty manner in order to focus on the goal of applying GenUTest to a large project. Some modifications were made to GenUTest in order to improve its performance and resource usage when it was used to capture and record JODE’s events during runtime. One of these modifications involved the selection of testable events. Rather than generating tests for all testable events, we modified GenUTest to only generate a test for the last event for each instance. This may provide some explanation to the differences in the code coverage measured obtained by the ‘system’ test and by the unit tests. GenUTest generated 462 unit tests, obtaining a code coverage of 4%, compared to a 24% coverage achieved by the regular run.

6.2 Capabilities and Limitations

GenUTest captures and records inter-object interactions during runtime. It supports the events: object instantiation, regular and static method-calls, read/write access of regular and static class fields. The recorded data is used to automatically generate unit tests and mock aspects. With the latter, classes can be tested in isolation, without having to modify their code. The tests generated by GenUTest support handling parameters and return values of primitive types, objects, and arrays.

Our experiments show that whereas GenUTest can be applied on middle sized Java projects (e.g. JODE), its employment on a wide range of projects is still not practical. This has sev-
eral reasons: GenUTest can not handle multi-threaded programs; supporting inner classes and anonymous classes is only partial; arrays are handled in an inefficient manner which does not scale very well. In addition, GenUTest consumes a lot of memory and its performance needs to be improved. In some cases it generates a huge amount of unit tests, some of which may be redundant.

6.3 Future Work

Some of the limitation of GenUTest could be overcome by further developing the tool. For example, to fully support a wide range of Java constructs such as inner and anonymous classes, JNI, the multi-threaded paradigm, etc.

Later, the comprehensiveness and quality of the generated tests should be improved. The former can be addressed by extending the testability criterion to also include events that do not necessarily return a value or throw an exception. This will lead to a better code coverage than the one obtained today. Also, GenUTest may generate lots of tests that contain many statements, some of which may be redundant. That is, there may be more than one test that exercise the CUT in the same manner, and this should be detected and removed. Also, a mechanism to reduce the number of statements in a test should be developed.

Another research direction focuses on the performance and scalability of the tool. Global repositories, such as the type repository and the event dependent repository, which contain information for all classes, must be divided accordingly. In order to effectively handle large arrays, the following should be implemented. First, arrays can be compressed before they are stored to disk. Second, an array should not always be stored in its entirety; in some cases, recording the changes that occur to an array is sufficient. The XML files should not contain redundant information and need to be more compact. This can be achieved by using shorter XML elements and node names. Also, records that are used numerous times can be replaced by references. When suitable, execution join points may be considered as a substitution to call join points. This will reduce the code bloat described earlier in Section 6.1.

It would also be interesting to study how the tool can be incorporated in the development process of a software project in maintenance. This can be performed in a controlled environment, e.g., an academic programming course. Students may be required to use GenUTest in their programming assignments and to report their experience. The effectiveness of GenUTest in locating regression bugs in existing projects can also be studied. The tool can first be used to generate unit tests for an old version of some software. Then, the unit tests can be executed with newer versions of the software. The discrepancies should be examined to determine if they uncover a regression bug.
Bibliography


