Int J Softw Tools Technol Transfer (2009) 11:273–290
DOI 10.1007/s10009-009-0115-4

REGULAR PAPER

GenUTest: a unit test and mock aspect generation tool

Benny Pasternak · Shmuel Tyszberowicz · Amiram Yehudai

Published online: 3 September 2009

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. One of the major corner stones of the agile development approach is unit testing. Agile methods require all software classes to have unit tests that can be executed by an automated unit-testing framework. However, not all software systems have 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 paper 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. Comprehensiveness of the unit tests depends on the software execution. We applied GenUTest to several open source projects such as NanoXML and JODE. We present the results, explain the limitations of the tool, and point out direction to future work to improve the code coverage provided by GenUTest and its scalability.

B. Pasternak · A. Yehudai
Tel-Aviv University, Tel Aviv, Israel

S. Tyszberowicz (✉)
The Academic College Tel-Aviv Yaffo, Tel Aviv, Israel
e-mail: tyshbe@tau.ac.il

1 Introduction

Unit testing plays a major role in the software development process. Extreme Programming (XP) [14] 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 [15].

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* [26] 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 refactor code safely, 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 refactor code whenever this can yield a better program structure.
2. **Simplifies integration:** Integrating units that have already been tested, eases the process.

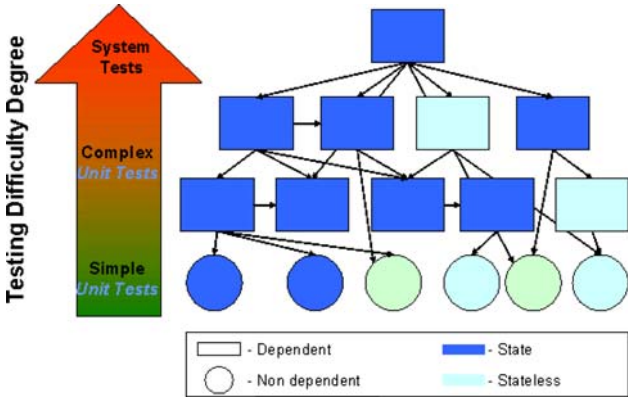

Fig. 1 A typical system unit dependency graph

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 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. A mock object is a unit testing pattern [6] that is classified under the category of simulation patterns. 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.

The number of unit tests for a given project may be very large. For instance, Microsoft reported that the code for unit tests is often larger than the code of the project [9]. In order to manage unit tests effectively, execute them frequently, and analyze their results, a unit testing framework should be used [14]. The framework automatically executes all unit tests and reports their results. One of the most popular unit testing frameworks is *JUnit* [17,23], 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 [37]. 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.

1.1 GenUTest

We have built GenUTest, a prototype tool that generates unit tests based on the proposed technique [32]. 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. Hence, a unit testing suite can be formed, assisting the development of the project using agile development methodologies.

Figure 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 [3,25].

The rest of this section provides a brief review of the AOP approach and a quick introduction to JUnit.

1.2 Aspect-oriented programming

Aspect-oriented programming is a programming paradigm which enables developers to develop software that maintains

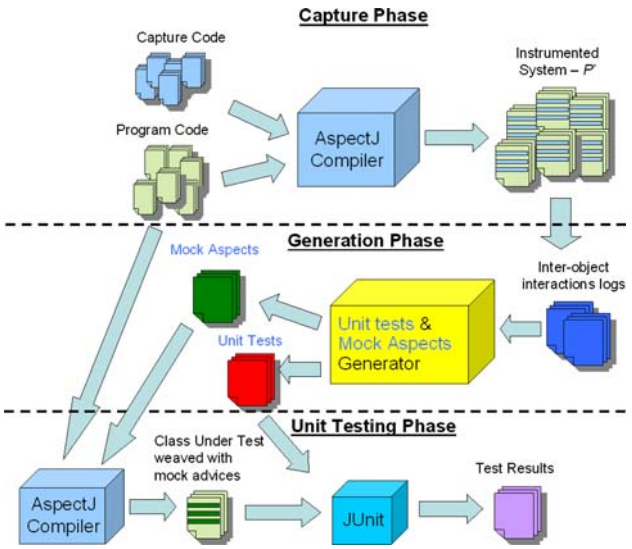

Fig. 2 GenUTest architecture

```
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;
    }
}
```

Fig. 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);
}
```

Fig. 4 A test for the add method

the important principle of Separation of Concern (SOC), specifically cross-cutting concerns [24]. SOC is the process of breaking a program into distinct features that overlap in functionality as little as possible. The procedural and the object-oriented programming paradigms provide constructs that support this process. However, SOC cannot 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 such a concern. The logging code typically 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*.

An *aspect* contains a set of *advices*, which are pieces of code to be invoked when specific events occur during execution of the program. These events are called *join points*, and include method calls, reading and modifying fields, handling exceptions, and more. A *pointcut* is a specification for a set of join points; for example, the pointcut `get(* C.x)` contains all join points in which the value of variable `x` of class `C` is read. There are three kinds of *advice*: *before advice* executes before the join point, *after advice* executes after it, and *around advice* replaces it completely (but can optionally invoke the original code).

1.3 JUnit

JUnit [17,23] was one of the first frameworks developed for unit testing in general and for Java programs in particular. It helped to create a standard way for writing unit tests. JUnit automates the execution of 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 Fig. 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 three and five is indeed eight. The code fragment is shown in Fig. 4.

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 throws an `ArithmeticException` as seen in Fig. 5.

Figure 6 shows the feedback given to the developer after running the tests. Note that we added another test that deliberately fails in order to show the outcome of a failing test as well.

```
@Test(expected=
      java.lang.ArithmeticException.class)
public void divideByZeroThrowsException()
{
    // code executing the test
    Calculator calc = new Calculator();
    int result = calc.div(3,0);
}
```

Fig. 5 A test that checks that an exception is thrown when dividing by zero

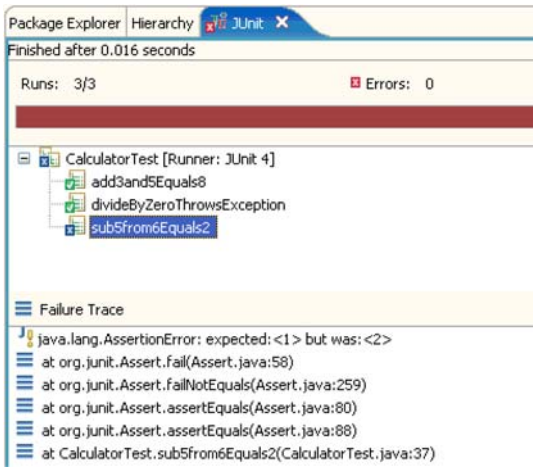

Fig. 6 Unit test results reported by JUnit

The rest of the article is organized as follows. In Sect. 2 we discuss related work. Section 3 describes the capture phase, and Sect. 4 explains how unit tests are generated from the captured data. Section 5 elaborates on the creation of mock aspects. In Sect. 6 we describe our experiments with the tool. We finally conclude with Sect. 7, where we summarize the capabilities and limitations of GenUTest and describe future research plans.

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 [1,8,29,31] are categorized 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 $f(A_1, A_2, \ldots, A_N)$, and the method returns R. In order to test this method, the tool needs to know how to construct values of the types $C, A_1, A_2, \ldots, A_N$. 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 $C, A_1, A_2, \ldots, A_N$. Using these inference rules, sequences of random method-calls can be generated automatically.

Symbolic execution tools, such as [5,38,40], 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. [13,30,33,42], capture and record method sequences, argument 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. These tools can also be used for the creation of test assertions. This is done by comparing the values obtained during the execution of the tests with the recorded values.

Test assertions can also be generated with the following approaches. Tools such as [8] 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, for example [5,9,31], infer an operational model of the CUT. This model consists of class invariants, and pre- and post-conditions for each method of the class. The operational model can be constructed either from user specifications or from dynamic analysis tools such as Daikon [36]. Daikon monitors the execution of the program under test and is able to infer class invariants, and pre- and post-conditions. The result of unit tests generated with techniques described earlier are then 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 sequences and the values captured in real executions. Thus, it is related to capture and replay tools. In [13,30,33,42] the capture process of Java programs is implemented by applying complicated instrumentation techniques on the target's bytecode. Those techniques 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 implement the capture mechanism in an effective, natural and elegant manner. Furthermore, it is easy to implement the tool for other AOP languages. Saff et al. [33], 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 [30], 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 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, as well as in [30] and in [33], are based on method-sequence representation. In this technique, the state of an object is represented using sequences of method invocations that produce the object. In the works described in [42] and in [13], 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 [42], 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 [13], 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 paper. 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, the method-sequence representation seems to be more suitable for unit testing (performing black box testing of the unit).

Note that there exist tools which are relevant to GenUTest but are not used for unit testing. Those tools are described in Sect. 3.1.

3 GenUTest: the capture phase

In the following two sections we describe the GenUTest tool. The current section 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.

3.1 Capture and replay

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 (GUIs). CAR tools, e.g. WinRunner [18], enable to record a sequence of GUI actions in the form of a test script. This 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.

Capture and replay tools can also record cross system interactions and inner software interactions. For instance, jRapture [34] is a tool for capturing and replaying the executions of Java programs during beta testing. It captures interactions between a Java program and the system, including GUI, file, and console inputs, etc. The captured data can be analyzed by testing personnel. The process provides feedback and information that cannot be provided by regular feedback and bug reports submitted by beta testers. Tools such as SCARPE [30] capture and replay inner software interactions of Java applications. 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.

There are various approaches to implement CAR tools. We focus on instrumentation techniques used to build such tools. Instrumentation is a widely used technique that changes a program in order to modify or to extend its behavior. For example, profiling tools such as VTune [20] instrument a given program with special code. When the instrumented program is executed, performance parameters are measured and recorded. The recorded data are analyzed by the tool to assist developers in finding performance bottlenecks.

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.

3.2 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, structured, and an intuitive method for instrumenting code.

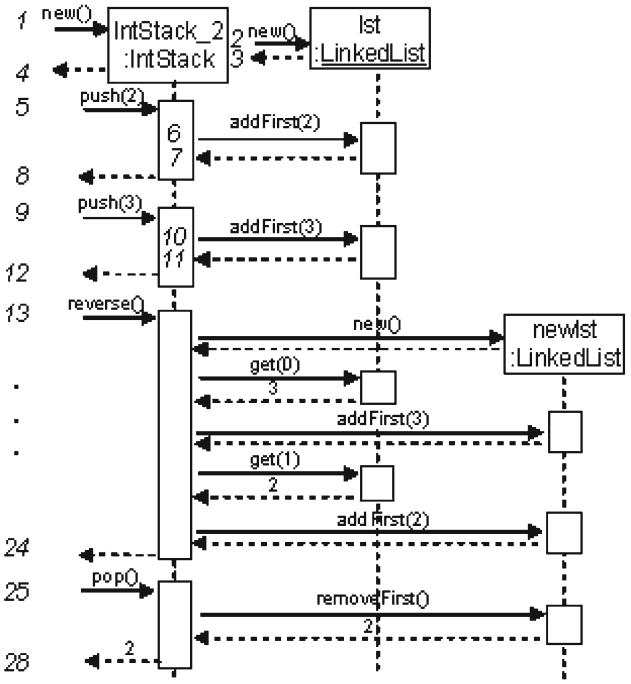

Fig. 7 A sequence diagram describing a scenario of the stack behavior

Our approach makes it easy to implement the tool for other AOP languages as well.

To illustrate our ideas we employ a simple example which we use throughout the article. This is an integer stack implemented using a linked list. Besides the conventional stack operations, the stack also supports a *reverse* operation, which reverses the order of the items in the stack. Figure 7 presents a UML sequence diagram which describes a possible scenario of the stack behavior.[1]

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. In the rest of the article we refer to constructor-calls, method-calls, and read/write field accesses as events. The above-mentioned join points can be matched using a static approach or a dynamic one.

[1] The numbers in *italic* are used to denote event intervals which are introduced in Sect. 3.2.2. 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.

3.2.1 The static approach

In this approach, we statically analyze the code of the target program in order to obtain information regarding the classes and their respective constructors, methods and fields. 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.

3.2.2 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 in a general manner that enable them to match all the required events. However, the tradeoff of utilizing such 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 pointcuts in order to match those events. For instance, to match all the public constructor-calls, we define the following pointcut:

```
pointcut publicConstructorCall():
    call(public *..new(..)) &&
    !within(GenUTest.*);
```

The following pointcut declaration matches all the public method-calls:

```
pointcut publicMethodCall():
    call(public * *(..)) &&
    !within(GenUTest.*);
```

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

```
pointcut publicSetter():
    set(public * *..*) &&
    !within(GenUTest.*);
```

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

```
pointcut publicGetter():
    get(public * *..*) &&
    !within(GenUTest.*);
```

The `!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. The advices also have to record the execution order of the events, and this can be achieved by time stamping the events. This, however, is not enough to fully describe the execution relationships of two different events. Let us examine, for instance, the events `push(2)`,

`addFirst(2)`, and `push(3)` shown in Fig. 7. Suppose the time stamp assigned to an event is simply a counter which is incremented by one each time an event begins execution. If t is the time stamp assigned to `push(2)` as it begins, then the next event is `addfirst(2)`, which begins at $t + 1$, and $t + 2$ is assigned to `push(3)`. Events might be assigned a different time stamp if time is incremented and assigned at the end of an event. In our example, `addfirst(2)` is the first event to finish execution, and it is assigned the time stamp t. Then $t + 1$ is assigned to `push(2)` and $t + 3$ to `push(3)` (note that before it finished, `addFirst(3)` ended at $t + 2$). Neither of the timestamp sequences correctly portray the execution of the events `push(2)` and `addFirst(2)`. This is due to the fact that the event `addFirst(2)` occurred *during* the execution of `push(2)`.

Therefore, we represent time by a global sequence number which is incremented *before* an event begins execution **and** *after* it finishes execution. The interval [before, after] is called the *event interval*. Using the *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. This enables GenUTest to record the time that the event starts execution, to execute the event, to record the event, and to record the time it ends.

Capturing an event involves recording its 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 example, the attributes of the method `IntStack_2.push(3)` (cf. Fig. 7) are: the AspectJ signature, which consists of the full name of the method (`IntStack_2.push()`), the arguments types (`int`), the access modifier (`public`), 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 (*thisJoinPoint*). This construct provides access to the following methods:

1. **getArgs()**: Returns an array of objects (`Object[]`) which can be traversed in order to explore the values of the arguments passed to methods and constructors.
2. **getSignature()**: 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 before.
3. **getThis()**: For advices that are used at call join points, as GenUTest does, the method `getThis()` returns a reference to the instance invoking the event. For example, let us examine the method-call `addFirst(2)` invoked during the execution of the method `push(2)`, as shown in Fig. 7. The advice is executed at the call location. Thus, `getThis()` will return a reference to the object `IntStack_2`. In case an event is invoked within a static method, `getThis()` returns null.
4. **getTarget()**: Similar to `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 `lst`. In case a static method is invoked or a static field is accessed, `getTarget()` returns null.

3.3 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 the capture phase and to distinguish between them. An object record is comprised of three fields: *object ID*, *static type ID*, and *dynamic type ID*.

3.3.1 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 [19], 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 == null ? k2 == 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.

3.3.2 Static and dynamic type IDs

The dynamic type ID is obtained by querying the *type repository* which maps types to IDs. The type repository obtains the class of the object using the `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:

```
<MethodCall>
 <returnValue class="PrimitiveTypeValue">
   <value>2</value>
   <typeID>6</typeID>
 </returnValue>
 <returnVarName>Integer_11</returnVarName>
 <eventInterval>
  <before>37</before>
  <after>42</after>
 </eventInterval>
 <signatureID>12</signatureID>
 <objectRecord>
  <instanceID>2</instanceID>
  <staticClassID>13</staticClassID>
  <dynamicClassID>13</dynamicClassID>
 </objectRecord>
</MethodCall>
```

Fig. 8 An XML representation of a Method-Call event

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.

3.4 The 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 to type records. These will be discussed briefly.

The serialization is achieved using the XStream serialization library [41]. The library supports the serialization of complex nested objects which is especially useful when serializing arrays. Figure 8 presents an XML fragment of a serialized method-call. The *returnValue* element records the return-value of the method and the ID of the type record. The latter, which describes the return value type, 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* represents the appropriate event interval (using the before and after values). 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_2 in the example), and the IDs of its static and dynamic types.

3.4.1 The 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 the 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 encode various properties such as transient, public, static, volatile, synchronized, etc.

The records 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>
   <modifiers>1</modifiers>
  </MethodSignatureParser>
</entry>
```

3.4.2 The type record

A type record contains information about a type in the program *P*. It is stored in the *type repository*, and has 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*: Is the type a primitive?
5. *Array Indicator*: Is the type an array?
6. *Component Type ID*: The elements type of an array.
7. *Comparison Indicator*: Is `equals` applicable to this type?
8. *Modifier*: The Java access modifiers of the type.

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

```
<entry>
  <int>13</int>
  <TypeRecord>
   <typename>IntStack</typename>
   <isPrimitive>false</isPrimitive>
   <modifiers>1</modifiers>
  </TypeRecord>
</entry>
```

3.5 Special cases

3.5.1 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 using a constructor call. This fact is discussed in detail in Sect. 3.5.2. Iterator objects are another example for objects whose creation is implicit. They 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.

When the event-dependent objects are arrays, additional handling required. 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 to an object of type `Object[]` and the other is to an object of the non-primitive type. GenUTest cannot 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. `someRef = someObj`
2. `someRef = someObjArr[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, the process is applied in a recursive manner.

3.5.2 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 (=):

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

which is equivalent to:

```
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" [35]. When an assignment is used, a String constructor is implicitly called.

This implicit constructor-call cannot 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 cannot 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 to the event logger, which in turn records it as if it were a regular constructor event. The event logger cannot distinguish between a real String constructor event and a synthetic one.

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

3.5.3 Class object

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

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

Class objects can also be obtained using the Reflection API [21]:

```
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 cannot be accessed by AspectJ, and therefore it cannot be accessed by GenUTest either. Thus, GenUTest cannot capture the initialization of the `class` field, and when the reference 'c' is encountered, GenUTest cannot 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.

4 GenUTest: the unit test generation phase

In this section, 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. Fig. 9).

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 *restoreObjectState* procedure selects the method-calls which are needed to execute the test, whereas

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

Fig. 9 Unit test generated for pop() method-call

generateStatement generates the 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 3–7 in Fig. 9) 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* [39]. 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 $State_T(obj) =< e_{t_1}, e_{t_2}, \ldots, e_{t_n} >$, where $e_{t_1}, e_{t_2}, \ldots, e_{t_n}$ are all the events that occurred on the object *obj* at times $t_1 < t_2 < t_3 < \cdots < t_n <= T$ and t_1 being the creation time of *obj*.

In order to test *obj* at its correct state, all the events invoked on *obj* prior to T need to be reinvoked. Suppose 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 Fig. 7. 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 occurred before time stamp 13, must be reinvoked.

Note that the creation event is either a constructor event or a method event applied to another object, and it 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:

```
<return variable> = (<return type>)
        <object ref>.<method name>
              (<arg #1,..., 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 are then surrounded by a pair of quotes.
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:
   `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:

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

The arguments are processed in the same manner as parameters are processed when generating a method-call Java statement.
For a read field access event the procedure generates a statement of the following form:

```
<variable> = <object ref>.<field name>;
```

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

```
<object ref>.<field name> = <arg #1>;
```

where `arg #1` is a single argument in an argument list and is handled similarly. If the accessed field is a static one, the object reference is the name of the class.

Method Interval	obj1	obj2	obj3	...
[1,2]	obj1 = new Type1()			
[3,4]			obj3 = new Type3()	
[5,8]		obj2 = new Type2()		
[9,20]			obj3.initialize()	
[21,30]		obj2.goo1(obj3)		
[31,50]	**obj1.foo1(obj2)**			
[51,64]		obj2.goo2()		
[65,80]	obj1.foo2()			
...				

Fig. 10 Method-calls invoked on the objects obj1, obj2, and obj3

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

```
<object ref1>=<array ref>[<array idx>];
```

4.1.3 An example

In the following example, we demonstrate how both procedures work on a more complicated example involving objects.

Figure 10 presents the method-calls occurring at consecutive event intervals for three different objects: `obj1`, `obj2`, and `obj3`. Suppose that GenUTest encounters the method-call `obj1.foo1(obj2)` which occurred at time stamp 31. In order to invoke the method-call, GenUTest must restore the target object `obj1` to its correct state at time stamp 31. This is achieved by the procedure *restoreObjectState* which selects the constructor-call `obj1 = new Type1()` that occurred at time stamp 1. The procedure *generateStatement* is then invoked in order to generate the Java statements 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 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 Fig. 11.

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)`, the statements at lines 2 and 4 in Fig. 11 would be generated twice. This leads to an incorrect sequence of statements which in some cases

```
1 Type1 obj1 = new Type1();
2 Type3 obj3 = new Type3();
3 Type2 obj2 = newType2();
4 obj3.initialize();
5 obj2.goo1(obj3);
6 obj1.foo1(obj2);
```

Fig. 11 Statements generated to restore the correct state of obj1 and obj2

might affect the state of the objects. The post-processing task detects and disposes of such redundant statements.

4.2 Step II: test assertion

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

4.2.1 Case I: a value is returned

In this case GenUTest generates 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 compared directly using one of JUnit's assert statements. In the example, the return variable intRetVal6 ($value_{test}$) is compared to 2 ($value_{captured}$) (cf. Fig. 9, line 10).

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 Sect. 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 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 GenUTest to discern information about the objects' fields. The discovered fields are compared based on their types, using the `CompareToBuilder` class of the *Commons Lang* [7] library.

Since the `CompareToBuilder` class does not support array comparison, a slightly different approach is taken in the case of arrays. If the arrays contain primitive types or strings, the `Arrays.equals` method [2] is used to perform the comparison. Otherwise, GenUTest provides a special method. This method accepts as input two objects to be compared. It first ensures that both objects are arrays of the same length. It then recursively traverses both arrays, performing a deep comparison.

```
1 @Test(expected=
2     NoSuchElementException.class)
3 public void testpop2() {
4  // test execution statements
5  IntStack IntStack_3 = new IntStack();
6  IntStack_3.pop();
7 }
```

Fig. 12 Unit test generated for exception throwing pop() method-call

4.2.2 Case II: an exception is thrown

In case 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: `@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 12 presents the generated code for this scenario.

5 Generating mock aspects

This section describes what mock objects are. We explain the virtual mock pattern and mention the benefits of mock object frameworks. We then introduce a new concept, namely the 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, sometimes requiring the developer's intervention. A common approach to deal with this issue is the use of mock objects [26].

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 should replace 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).

```
// 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);
```

Fig. 13 Object creation using the factory pattern

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 exist yet.
5. It will contain data and/or methods which are needed only for testing purposes.

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 object creation, the CUT's code must be modified as follows. Code fragments which explicitly create instances of a real object must be modified 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 the *Abstract Factory Pattern* [16]. 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 13 shows a code snippet that creates an object using the pattern.

5.1.1 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* [27] 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 [10]) 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(<ret-val>)`. In those statements, `<method-call>` denotes the expected method-call and `<ret-val>` 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 14 demonstrates how the List object is mocked using EasyMock in the test `testpop1` (refer to Figs. 7, 9). The use of EasyMock required modifying the code of `IntStack` 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 called 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 15 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. Fig. 7), the incoming method-calls of `IntStack` are: `new IntStack(), push(2),`

```
1  public void testpop1WithEasyMock() {
2   // step #1
3   MyAbstractLinkedList mock =
4    createMock(MyAbstractLinkedList.class);
5
6   // step #2
7   mock.addFirst(2);
8   mock.addFirst(3);
9
10  // calls of IntStack_2.reverse to List
11  expect(mock.createNewInstance()).
12     andReturn(mock);
13  int[] arr = new int[] { 3,2 };
14  for (int i = 0; i < 2; i++) {
15      expect(mock.size()).
16         andReturn(2);
17      expect(mock.get(i)).
18         andReturn(arr[i]);
19      mock.addFirst(arr[i]);
20  }
21
22  expect(mock.size()).
23     andReturn(2);
24  expect(mock.removeFirst()).
25     andReturn(2);
26
27  // step #3
28  replay(mock);
29
30  // step #4
31  IntStack IntStack_2 = new IntStack(mock);
32  // test code
33  IntStack_2.push(2);
34  IntStack_2.push(3);
35  IntStack_2.reverse();
36
37  // test assertion
38  int intRetVal6 = IntStack_2.pop();
39  assertEquals(intRetVal6, 2);
```

Fig. 14 Testing pop with the use of the EasyMock framework

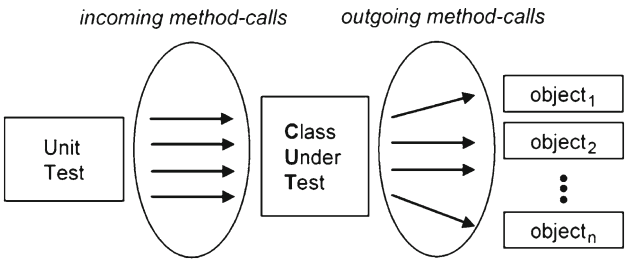

Fig. 15 A Class Under Test (CUT) and its incoming and outgoing method-calls

`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 $object_i$, $1 \leq i \leq n$, 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 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 $[\text{before}_A, \text{after}_A]$ of method-call $A()$, where before_A identifies the beginning of $A()$'s execution, and after_A identifies the end of execution.

Definition 3 Method-call $A()$ *contains* method-call $B()$ if $mi(A())$ contains $mi(B())$, that is,
$[\text{before}_A, \text{after}_A] \supset [\text{before}_B, \text{after}_B]$.

Following these definitions, we observe that:

1. Method-call $B()$ resides within the control flow of a method-call $A()$ iff method-call $A()$ contains method-call $B()$.
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 $Outgoing(I())$ is the ordered sequence

$$\langle Io_1(), Io_2(), \ldots, Io_n() \rangle,$$

where $Io_1(), Io_2(), \ldots, Io_n()$ are all the chronologically ordered outgoing method-calls contained in the incoming method-call $I()$.

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

Figure 7 displays four outgoing method-calls of the method `addFirst()`. The first one, `addFirst(2)`, is contained in the incoming method-call `push(2)`. Thus, $mi(push(2))$ is [5, 8], $outgoing(push(2))$ is $\langle addFirst(2) \rangle$, and the pair ([5, 8], 1) uniquely identifies `addFirst(2)`. Similarly, the second outgoing method-call `addFirst(3)` is uniquely identified by the pair ([9, 12], 1). The other outgoing method-calls, `addFirst(3)` and `addFirst(2)`, are both contained in the incoming method-call `reverse()`. Hence, $mi(reverse())$ is [13, 24] and $outgoing(reverse())$ is $\langle new, get(0), addFirst(3), get(1), addFirst(2) \rangle$. The outgoing method-calls `addFirst(3)` and `addFirst(2)` are uniquely identified by the pairs ([13, 24], 3) and ([13, 24], 5), respectively.

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

```
@Test public void testpop1() {
    IntStackMockAspect.setMI(1,4);
    IntStack IntStack_2 = new IntStack();
    IntStackMockAspect.setMI(5,8);
    IntStack_2.push(2);
    IntStackMockAspect.setMI(9,12);
    IntStack_2.push(3);
    IntStackMockAspect.setMI(13,24);
    IntStack_2.reverse();
    IntStackMockAspect.setMI(25,28);
    int intRetVal6 = IntStack_2.pop();
    assertEquals(intRetVal6,2);
}
```

Fig. 16 Unit test for pop() method-call with helper statements

- Know all the incoming method-calls of the CUT.
- Keep track of the outgoing method-calls sequence for each incoming method-call.

The mock aspect generation algorithm works as follows:

1. It calculates $outgoing(I())$ for each incoming method-call $I()$ of the CUT.
2. It uniquely identifies each outgoing method-call.
3. It creates the mock-aspect code.
   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 $restoreObjectState$ described in Sect. 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 Fig. 16.

Figure 17 shows a code snippet of the mock aspect for the CUT `IntStack`. This code mimics outgoing method-calls to the method `get()`.

6 Experiments

During the development of GenUTest, we have applied it to some small programs. Later, as the tool supported more features and became more robust, larger examples were used. Two types of experiments were carried out with the tool: coverage and scalability.

```
// 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 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;
      }
   }
}
```

Fig. 17 A code snippet of the mock aspect generated for IntStack

6.1 Evaluating coverage

We performed coverage experimentation on four programs. The course of the coverage experiment for each program P is illustrated in Fig. 18. Let R denote some run of P. First we have to instrument P with GenUTest's capture code. The instrumented program P' is then executed, and the events are captured and logged into L. When the execution of P' ends, GenUTest processes L and generates unit tests and mock aspects for each class. During the experiments we modified GenUTest in order to produce two kinds of unit tests: (a) U: Standard unit tests which are generated for every testable event (i.e., an event that either returns a value or throws an exception), and (b) U': Unit tests which are generated for *every* event. The test assertion for events that neither return a value nor throw an exception is the trivial one: assertTrue(true). To make this a real unit test we need to compare the state of the object at runtime with its state after executing the test; see also Sect. 7.

U and U' are executed using JUnit and then 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 two sets of unit tests—U and U'. Let us denote the code coverage of the run R by RCC, the code coverage of U (the actual code coverage) by ACC, and the code coverage of U' (the potential code coverage) by PCC. The coverage is measured using the EclEmma [11] Eclipse plugin [12].

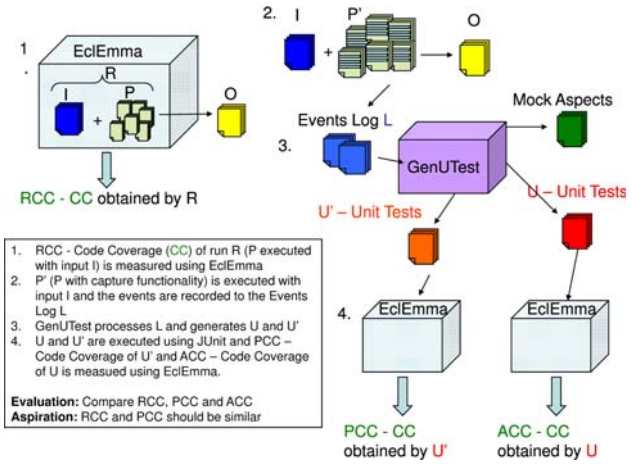

Fig. 18 Illustration of the coverage experiment

	RCC	PCC	ACC
IntStack	81.3%	81.3%	81.3%
NanoXML	19.3%	19.3%	15.2%
ATM	14.4%	14.4%	14.3%
JODE	24%	24%	4.7%

Fig. 19 A table displaying results of coverage experiments

Ideally, RCC and PCC are similar. We revisit the IntStack example in order to explain why PCC and ACC might be different. Suppose the last operation in Fig. 7 is reverse [13,24] and not pop [25,28]. In this case there are no testable events in the sequence when GenUTest generates U, and therefore no unit tests will be generated—leading to an ACC of 0%. When GenUTest generated U', a unit test is generated for every event (regardless whether it is testable or not), obtaining the code coverage PCC which is similar to RCC. The issue of narrowing the gap between ACC and PCC is addressed in Sect. 7. The coverage-experiments results are summarized in Fig. 19.

First, we applied GenUTest on the IntStack example described in Sect. 3.2. A module test was designed to examine the IntStack class using a scenario based on the sequence diagram described in Fig. 7. GenUTest generated two unit tests for the IntStack class. In this simple example, the unit tests obtained the same coverage as the module test. Note that the coverage is only 81.3% since the methods empty() and top() have not been invoked.

Then, we applied GenUTest on NanoXML [28], a small open source project. 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 19.3%. When instrumented with GenUTest 111 unit tests have been generated. Execution of the unit tests achieves code coverage of 15.2%. The 4.1% gap is mainly due to the fact that not all events return a value or throw an exception. When the testability criterion in GenUTest is loosened, e.g., when every event is tested, GenUTest generates 124 unit tests, and the PCC is 19.3%, identical to RCC, the coverage of the run without tests.

Another project we experimented with is a single threaded ATM simulation [42]. The project consists of approximately 40 classes and about 2,100 lines of code. Execution of the system test provided with the project obtained code coverage of 14.4%. GenUTest generated 15 unit tests, achieving code coverage of 14.3%. When unit tests are created for all events, the code coverage is 14.4%, and 24 unit tests are created. In this experiment, GenUTest was able to generate test cases for almost all the scenarios executed during the system test.

6.2 Evaluating scalability

For the scalability experiment, we have applied GenUTest to a medium sized project called Java Optimize and Decompile Environment (JODE). JODE is an open source project which spans approximately 35,000 lines of code [22]. We used JODE to examine how GenUTest handles larger programs with a huge amount of events. This turned out to be a non-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 64 K 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 Fig. 3. The execution consumed a lot of memory and took a lot of time (around 30 min). It also uncovered some bugs and errors in GenUTest. Some modifications were made to GenUTest to improve its performance and resource usage when capturing and recording 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 of each instance. This may provide some explanation to the differences in the code coverage obtained by the system test and by the unit tests. GenUTest generated 510 unit tests, obtaining code coverage of 4.7%, compared to the 24% coverage achieved both by the regular run and when unit tests have been created for all events by the run (PCC), yielding 548 unit tests.

7 Conclusion

In this section, we briefly describe the efficacy and current restrictions of GenUTest and discuss some ideas for future research.

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 are 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 to medium sized Java projects (e.g., JODE), its employment on a wide range of projects is still not practical. This has several reasons: GenUTest cannot 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. We need to further improve GenUTest to handle those issues as well as others. In some cases GenUTest generates a huge amount of unit tests, some of which may be redundant, i.e., there may be more than one test that exercises the CUT in the same manner. These can be removed using the techniques described in [39].

The comprehensiveness and quality of the generated tests should be improved. The former can be addressed by extending the testability criterion to include also events that do not return a value or throw an exception. During the capture phase, the post-event object state can be stored. This runtime state can then be compared to the object state in the unit test. This should reduce the current gap between the actual and the potential code coverage.

One of the major limitations of GenUTest is that the created unit tests depend on the captured test run. As we saw in our experiments, the coverage of this run, and therefore of the unit test created by GenUTest as well, may be low. In order to achieve a better coverage of the captured run we plan to use the KeY system [4]. KeY generates JUnit tests that achieve a high code coverage. However, KeY does not test units in isolation. It does not create mock objects, thus it executes all methods that are called by a tested method. The output generated by KeY will be provided as input to GenUTest. GenUTest will execute each test, and will generate a new JUnit test suite, where external methods that are called are replaced with mock aspects resulting in real unit tests. This combination of tools seems to promise the best of both worlds: the high coverage provided by KeY and testing modules in isolation using the unit tests created by GenUTest.

Another research direction focuses on the performance and scalability of the tool. In order to handle large arrays effectively, 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 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 [25] to reduce the code bloat.

It would also be interesting to study how the tool can be incorporated in the development process of a software project during its maintenance phase. 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 regression bugs.

Acknowledgments The authors would like to thank the anonymous referees for insightful comments and suggestions that helped to improve the paper.

References

1. Andrews, J.H., Haldar, S., Lei, Y., Li, F.C.H.: Tool support for randomized unit testing. In: Proceedings of the 1st International Workshop on Random Testing, pp. 36–45 (2006)
2. Arrays (Java 2 Platform SE 5.0).: http://java.sun.com/j2se/1.5.0/docs/api/java/util/Arrays.html. Accessed Jan 2009
3. AspectJ.: http://www.eclipse.org/aspectj. Accessed Jan 2009
4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. Lecture Notes in Computer Science, vol. 4334. Springer, Berlin (2007)
5. Boshernitsan, M., Doong, R., Savoia, A.: From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 169–180 (2006)
6. Clifton, M.: Advanced Unit Test, Part V—Unit Test Patterns, January 2004. http://www.codeproject.com/gen/design/autp5.asp. Accessed Jan 2009
7. Commons Lang.: http://commons.apache.org/lang. Accessed Jan 2009
8. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Softw. Pract. Experience **34**(11), 1025–1050 (2004)
9. d'Amorim, M., Pacheco, C., Xie, T., Marinov, D., Ernst, M.: An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), pp. 59–68 (2006)
10. Easy Mock.: http://www.easymock.org. Accessed Jan 2009
11. EclEmma.: http://www.eclemma.org. Accessed Jan 2009
12. Eclipse.: http://www.eclipse.org. Accessed Jan 2009
13. Elbaum, S., Chin, H.N., Dwyer, M.B., Dokulil, J.: Carving differential unit test cases from system test cases. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 253–264 (2006)
14. Extreme Programming.: http://www.extremeprogramming.org. Visited January (2009)
15. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (2000)

16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
17. Husted, T., Massol, V.: JUnit in Action. Manning Publications, Greenwich (2003)
18. HP WinRunner software.: http://www.hp.com. Accessed Jan 2009
19. IdentityHashMap (Java 2 Platform SE 5.0).: http://java.sun.com/j2se/1.5.0/docs/api/java/util/IdentityHashMap.html. Accessed Jan 2009
20. Intel VTune Performance Analyzer.: http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm. Accessed Jan 2009
21. Java programming dynamics, Part 2: Introducing reflection.: http://www-128.ibm.com/developerworks/java/library/j-dyn0603. Accessed Jan 2009
22. JODE (Java Optimize and Decompile Environment).: http://jode.sourceforge.net. Accessed Jan 2009
23. JUnit.: http://www.junit.org. Accessed Jan 2009
24. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), Jyväskylä, Finland (1997)
25. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications, Greenwich (2003)
26. Mackinnon, T., Freeman, S., Craig, P.: Endo-testing: unit testing with mock objects. In: Extreme Programming Examined, pp. 287–301. Addison-Wesley, Reading (2001)
27. Monk, S., Hall, S.: Virtual Mock Objects using AspectJ with JUnit, 2002. http://www.xprogramming.com/xpmag/virtualMockObjects.htm. Accessed January 2009
28. NanoXML.: http://nanoxml.cyberelf.be. Accessed Jan 2009
29. Oriat, C.: Jartege: a tool for random generation of unit tests for Java classes. In: Proceedings of the Second International Workshop on Software Quality (SOQUA), pp. 242–256 (2005)
30. Orso, A., Kennedy, B.: Selective capture and replay of program executions. In: Proceedings of the Workshop on Dynamic Analysis, pp. 1–7 (2005)
31. Pacheco, C., Ernst, M.: Eclat: automatic generation and classification of test inputs. In: Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP), Glasgow, Scotland, pp. 504–527(2005)
32. Pasternak, B., Tyszberowicz, S., Yehudai, A.: GenUTest: a unit test and mock aspect generation tool. In: Proceedings of Haifa Verification Conference, Lecture Notes in Computer Science, vol. 4899, pp. 252–266. Springer, Berlin (2007)
33. Saff, D., Artzi, S., Perkins, J., Ernst, M.: Automated test factoring for Java. In: Conference of Automated Software Engineering (ASE), pp. 114–123 (2005)
34. Steven, J., Chandra, P., Fleck, B., Podgurski, A.: jRapture: a Capture/Replay tool for observation-based testing. In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 158–167 (2000)
35. String (Java 2 Platform SE 5.0).: http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html. Accessed Jan 2009
36. The Daikon invariant detector.: http://groups.csail.mit.edu/pag/daikon. Accessed Jan 2009
37. van Vliet, H.: Software Engineering: Principles and Practice. 2nd edn. Wiley, New York (2000)
38. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 97–107 (2004)
39. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE), pp. 196–205 (2004)
40. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2005)
41. XStream 1.2.2.: http://xstream.codehaus.org. Accessed Jan 2009
42. Yuan, H., Xie, T.: Substra: a framework for automatic generation of integration tests. In: Proceedings of the International Workshop on Automation of Software Test, pp. 64–70 (2006)