Automatic Functional Testing of GUIs

L.A.Fitzgerald

Functional testing of GUIs can be automated using a test oracle derived from the GUI's specification and from a restricted set of randomised test data. As test data, a set of randomly distorted test objects seems to work well, especially starting as we do with a 'perfect' object and then distorting this more and more as the test progresses. The number of test cases needed seems to be much smaller than that reported in other random testing papers. More work is needed to see if the approach is generally applicable: if so, the test engineer can spend his time writing GUI specifications at a high level of abstraction, rather than hand-generating test cases.

1. INTRODUCTION

Software testing can be functional, structural, or random (Cai, 2005).

1. **Functional software** testing uses the functional specification of software to test the specified functions of software.
2. **Structural software testing** uses the internal details of software to generate test cases.
3. **Random software** testing uses a profile of the inputs that the software is expected to encounter in order to randomly generate test cases, and uses a Test Oracle to assess the correct operation of the software.

This paper discusses the automation of functional GUI testing using a test oracle derived from the GUI's specification and from a restricted set of randomised test data. There are three contributions:

- using JML specifications combined with run-time assertion checking to act as a Test Oracle for GUI programs. Leavens *et al* (Leavens, 2002), show how to combine formal interface specifications and a unit testing framework to produce a test oracle for unit testing. Here, the idea is applied to GUI testing.
- as test data, automatically generating a set of randomised Objects, each of which is generated from a basis object[1] which has been abstracted from the GUI's specification.
- demonstrating how runtime assertion checking and coverage analysis can be integrated with automated testing of GUI code.

These ideas were applied testing a simple GUI application which classifies triangles; its functionality is similar to that of classification programs which have been used in previous studies. The classifier has several methods that determine if specific parameters are instances of one of several types of triangles. Other approaches to testing the triangle classification program need hundreds of test runs to achieve good test coverage (Alzabidi, 2009) (Michael, McGraw and Schatz, 2001). The approach reported here requires tens of test runs to meet the same goal. This suggests that the use of randomised test objects as test cases is worthy of further consideration.

Tools used

JML

JML (Java Modeling Language) (Leavens, G.T. and Cheon, Y, 2006) is a formal behavioural interface specification language for Java, which allows one to specify both the syntactic interface of Java code and its behaviour. The behaviour of Java code which is what is of interest here, describing what should happen at runtime when the code is used. The behaviour of a method is specified using pre- and post conditions. Preconditions are not relevant for testing a GUI, which should be able to respond to any possible input.

[1] A *basis object* is one which can generate any object described by the GUIs input parameters.

jmlc

The JML compiler (jmlc), is an extension of a Java compiler and compiles Java programs annotated with JML specifications into Java byte-code (Leavens, G.T. and Cheon, Y, 2006). The compiled byte-code includes run-time assertion checking instructions that check JML specifications such as preconditions, normal and exceptional postconditions, and invariants

JUnit framework

JUnit is a Java framework that supports testing. JUnit features include:

- Assertions for comparing the outcome of tests with expected results
- Test Cases that exercise the target code
- Test fixtures that provide an appropriate environment for running the test cases
- Test suites which are collections of test cases
- Graphical and textual test runners for running the tests

JMLUnit

The jmlunit tool tool combines the JML compiler with JUnit (Leavens, G.T. and Cheon, Y, 2006). The tool uses JML specifications, processed by jmlc, to decide whether the code being tested works properly.

2. BACKGROUND

Automated random testing

Myers (Myers, 2004), gives a good, practical introduction to Software Testing, and includes an interesting discussion on testing the triangle classification program, variants of which are as an example used by several authors in the field, and also in this paper.

Sun *et al* (Jones, 2004), discuss the shortcomings of the capture/replay techniques used traditionally for testing GUI-Based Java Programs: “These techniques are marketed as labor saving tools for regression testing where the focus is ensuring that a later version exhibits the same behavior as an earlier version under the same set of stimuli. The utility of capture/replay techniques for testing during software development testing is questionable…” They present the case for “a specification-driven approach to test automation for GUI-based JAVA programs as an alternative to the use of capture/replay”.

Hamlet (Hamlet, 1994), gives an erudite background to Random Testing, emphasising the idea that the essence of Random Test is to be in a position to estimate the reliability of the program being tested over a range of input values. He makes the case that any attempt at modifying the generation of random data by, for example, taking a model of the software under test into account, makes the results less useful for estimating reliability. To be statistically significant, the number of such tests should be large, as a consequence of which, the presence of an automated Test Oracle is all but essential for Random Testing.

Chan *et al* (Chan, 2003), introduces the topic of Adaptive Random Testing, which “makes use of knowledge of general failure pattern types, and information of previously executed test cases, in the selection of new test cases”. The paper discusses the use of feedback to improve on Random Testing. The approach proposed by Cai *et al* (Cai, 2005), uses a stochastic process to generate test cases conforming to a targeted usage profile. The approach “treats software testing as a control problem, where the software under test serves as a controlled object that is modelled as

controlled Markov chain, and the software testing strategy serves as the corresponding controller". Another paper by Hu *et al* (Hu, 2008), addresses the topic, stating that Adaptive Test "consumes less test cases than Random Test."

Michael *et al* (Michael, McGraw and Schatz, 2001), discuss Genetic Algorithms (GA) in the context of test-data generation: "...the source code of a program is instrumented to collect information about the program as it executes. The resulting information, collected during each test execution of the program, is used to heuristically determine how close the test came to satisfying a specified test requirement. This allows the test generator to modify the program's input parameters gradually, nudging them ever closer to values that actually do satisfy the requirement. In essence, the problem of generating test data reduces to the well-understood problem of function minimization." They propose Genetic Search as a sophisticated technique for function minimisation. The example used in the paper uses same triangle classifier example as I use: " the standard GA has the best performance overall, covering about 93 percent of the code on average in about 8,000 target-program executions." Alzabidi *et al* (Alzabidi, 2009), investigate the performance of a proposed GA for path testing. They use an example triangle classifier similar to that used in this paper, and seem to generate an enormous amount of test data.

Murphy *et al* (Murphy Christian, Kaiser Gail, and Arias Marta, 2007), use equivalence classes of random data for testing: "Our data generation framework allows us to isolate or combine different equivalence classes as desired, and then randomly generate large data sets using the properties of those equivalence classes as parameters." This approach is close to that proposed in this paper, with the exception of not requiring a Test Oracle. The Equivalence Classes are also at a lower level of abstraction than the Object-based approach presented herein.

Chen et al (Chen, Shen, and Chang, 2008), propose "an 'object-based' approach, called component abstraction, to model the structure of a GUI. A GUI testing modeling language, GTML, is defined and a systematic approach in applying component abstraction is described." This Object-based approach is again similar to mine; however, the GUI testing modeling language approach diverges from the approach I use.

GUI test implementation

There is a large number of papers which deal with implementation issues such as: JUnit, JML, EMMA, Abbot and Ant. Among some of the more useful papers are: Verzulli (Verzulli, 2003), which introduces JML and some of its most important declarative constructs and Schneider (Schneider, 2000), which deals with techniques for building resilient, relocatable, multithreaded JUnit tests.

Wall (Wall, 2008) presents a guide to getting started with the Abbot Java GUI Test Framework, and Roubtsov (Roubtsov, 2006) gives a Step-by-Step Introduction to the EMMA coverage toolkit. Hatcher and Loughran (Hatcher, 2007), provide all you need to write an Ant script.

Leavens (Leavens, G.T. and Cheon, Y, 2006) gives a good overview of the JML modelling language, while Breunesse and Poll (Poll, 2003) deal with JML specifications with model fields. This latter subject (model fields) is, in my view, the key to making JML specifications tractable.

Cheon and Leavens' paper on JML and JUnit (Leavens, 2002) is of particular importance. They present "a simple but effective approach to implementing test oracles from formal behavioural interface specifications." The specifications are pre- and post-conditions written in JML.

Finally, I would never have gotten off the ground without the help of Mark Sebern's web-site (Sebern, 2008-2009). His very clear instructions for tool installation are especially helpful.

AUTOMATIC RANDOM TESTING OF GUIS

Work outline

The work included the following steps:

1. Using the Abbott API to inject random data into a program under test, (Tridentify and Triangle classes) with JML pre/post conditions added to an action handler.
2. Generation of test data based on Boundary Value and Equivalence Partition tests, using these to validate the JML pre/post conditions and to measure test coverage.
3. Generation of object-based random test data, using these with the JML postconditions to test the GUI.
4. Comparison of results and generation of documentation

Program under test

The GUI under test is a graphical front end for a triangle classification program.

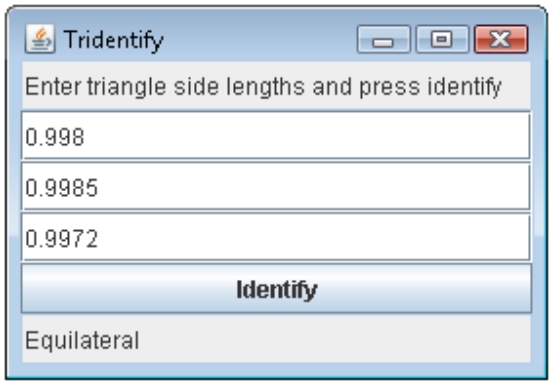

Figure 1 - GUI under test

The specification for the triangle classification program is:

```
* More precise version of specification:
*
* 2. When the use presses the "Identify" button, display the output
*    in the output text field using the following rules
*
* Rule  Inputs                              Output
* ----  ----------------------------------  --------------------------
*   1   any of x,y,z are invalid doubles    "Invalid input"
*
*   2   x,y,z all valid text for doubles,   "Invalid triangle"
*       but x,y,z form an invalid triangle
*
*   3   x,y,z all valid text for doubles    "Scalene"
*       and x,y,z form an valid triangle
*       and x!=y!=z
*
*   4   x,y,z all valid text for doubles    "Equilateral"
*       and x,y,z form an valid triangle
*       and x=y=z
*
*   5   x,y,z all valid text for doubles    "Isosceles"
*       and x,y,z form an valid triangle
*       and any two of the sides are equal
```

Figure 2 - “More precise version” of specification

This specification in fact defines the behavior of the GUI.

Hand-generated test data

A set of hand-generated test data was used to validate Tridentify.showAnswer's JML Specification. Some of the test data were derived from the Rules in the “more precise version” of the specification for Tridentify. The remainder were derived from a document (Brown, 2008) which describes the errors which were inserted into the Triangle class.

Auto-generated test data

It is not feasible to generate a set of integer 3-tuples, apply them to the GUI input fields and expect to achieve 100% code coverage in a short time, especially when the time delays associated with exercising the actual GUI are taken into account. The application of floating-point 3-tuples compounds the problem.

If the development process which produced the software to be tested has included a requirements engineering phase then the outputs of the Use-Case Analysis will include a specification of one or more objects which are parameterised by the fields in the GUI. It makes sense therefore, to generate not a set of random numbers to be applied on an *ad-hoc* basis to the fields of the GUI, but a set of randomised Objects, each of which is based on a basis object which has been abstracted from the GUI’s specification.

It is natural to select an equilateral triangle as the basis object for testing our example GUI, and generate a set of triangle objects by adding initially, a little noise to the parameters describing the object, and then, as the test progresses, adding more and more noise. It is useful to anticipate the experimental setup section below, and describe how this was done:

Two sub-sets of random data are generated: the first being divided by 256 in order to provide double data, and the second used to provide integer data. In each case the range of the random number generated starts out small and increases exponentially as the test progresses. Thus the character of the test data depends heavily on the number of tests used. The numbers generated are used to add 'noise' to the lengths of the sides of a basis triangle to produce triangle objects. These triangles are input into a queue; the data output from the queue are normally taken from its head, but may be taken from other elements with a probability which decreases as element's index in the queue increases. So, if a particular triangle appears in the test stream, it is likely to be repeated.

Pseudocode for generating the first set of random data is presented in Figure 3 below:

```
function generateStreamOfRandomTriangles()
        randRangeIncr = Integer.MAX_VALUE)/(3.0*nTests);
        randRangeExp  = 1.0
        construct a baseTriang with sides of length: x = 1.0; y = 1.0; z = 1.0;

        for(i = 0 to nTests/2)
                getDistortedTriangle()
                add it to a First-in, random-out (FIRO) queue of Triangles.
                remove a triangle from the queue
                // this removes elements from the queue from a random location
                // in the queue with the following probabilities:
                // Queue head (highest probability)
                // position 1 (lower probability)
                // ...
                // Queue tail (lowest probability)
                put the triangle in the output stream
        end
end

function getDistortedTriangle()
   return a new baseTriang with sides of length:
        x * (1.0 + genRandDbl())
        y * (1.0 + genRandDbl()) and
        z * (1.0 + genRandDbl())
end


function genRandDbl()
        generate a random number in the range [0 ..randRange]
        subtract randRange/2.0 and divide the result by 256
        store the result in retVal
        randRange = e^randRangeExp
        randRangeExp += randRangeExp + randRangeIncr;
        return retVal
end
```

Figure 3 - Pseudocode for generating the first set of random data

Using JML to provide a "test oracle" for GUI programs

Gary Leavens' JML Implementation Documentation web page states that "The idea behind jmlunit is to use JML's runtime assertion checker as a test oracle and use JUnit as a testing framework. The generated test classes send messages to objects of the Java classes under test; they catch assertion violation exceptions from test cases that pass an initial precondition check. Such assertion violation exceptions are used to decide if the code failed to meet its

specification, and hence that the test failed. If the class under test satisfies its interface specification for some particular input values, no such exceptions will be thrown, and that particular test execution succeeds. So the automatically generated test code serves as a test oracle whose behaviour is derived from the specified behaviour of the target class."

In order to use JML to provide a "test oracle" for GUI programs, all that is necessary is to instrument the objects which are invoked directly from the GUI code with JML postconditions.

How model variables work

A *model* field is a specification-only field for holding an abstraction of program data.

A *represents* clause describes how model fields can be computed from actual fields

An *ensures* clause specifies a method's postcondition. It may refer to the method's parameters and to its result value. Note that pre-conditions aren't relevant for a GUI specification, since a GUI should be able to handle any input value.

As an example, take one of the *model* fields from Figure 4:

```
model boolean validParameters;
```

The *represents* clause for this is:

```
represents validParameters  <- !(    (xParameter == null)      || (yParameter == null)      || (zParameter == null)
                                  || (xParameter.isInfinite())|| (yParameter.isInfinite())|| (zParameter.isInfinite())
                                  || (xParameter.isNaN())     || (yParameter.isNaN())     || (zParameter.isNaN())    );
```

Consider also the *model* field:

```
model String answerString;
```

and its *represents* clause:

```
represents  answerString <- tF_answer.getText();
```

Now one can see that the first line of the *ensures* clause is at quite a high level of abstraction:

```
ensures  (!validParameters) && answerString.equals("Invalid input") ...
```

At this high level of abstraction, it's easy to see the correspondence between the specification for showAnswer in Figure 2 and the *ensures* clause in Figure 4.

Even for a simple example like this, writing a JML GUI specification would be all but impossible, due to its complexity, without the use of model variables. These latter allow one to effect a hierarchical decomposition of the JML specification, which greatly eases the problem of writing a JML GUI specification.

None of this would be possible, if the triangle classifier had not come with a well-written informal specification. This has important implications for the "design for testability" of GUI applications. One can't test something which doesn't have clear specifications.

```
JML Spec:
  model String xParameterString;
  model String yParameterString;
   model String zParameterString;
   model String answerString;
   represents xParameterString <- tF_x.getText();
   represents yParameterString <- tF_y.getText();
   represents zParameterString <- tF_z.getText();
   represents answerString <- tF_answer.getText();
   model DoubleValidator doubleValidator;
   represents doubleValidator <- DoubleValidator.getInstance();
   model Double xParameter;
   model Double yParameter;
   model Double zParameter;
   represents xParameter = doubleValidator.validate(xParameterString);
   represents yParameter = doubleValidator.validate(yParameterString);
   represents zParameter = doubleValidator.validate(zParameterString);
   model boolean validParameters;
   represents validParameters  <- !(  (xParameter == null)    || (yParameter == null)    || (zParameter == null)
                        || (xParameter.isInfinite())|| (yParameter.isInfinite())|| (zParameter.isInfinite())
                        || (xParameter.isNaN())     || (yParameter.isNaN())     || (zParameter.isNaN()));
  model double xValue;
  model double yValue;
  model double zValue;
  represents xValue <- (validParameters) ? xParameter.doubleValue() : 0;
  represents yValue <- (validParameters) ? yParameter.doubleValue() : 0;
  represents zValue <- (validParameters) ? zParameter.doubleValue() : 0;
  model boolean nonNegativeValues;
  represents nonNegativeValues <- (xValue >= 0) && (yValue  >= 0) && (zValue  >= 0);
  model double longest;
  represents longest <- (validParameters && nonNegativeValues) ?
Math.max(Math.max(xValue,yValue),Math.max(xValue,zValue))  : 0;
model boolean validTriangle;
represents validTriangle <- (2 * longest) <= (xValue + yValue + zValue);
model boolean noneEqual;
represents noneEqual <- ((xValue != yValue) && (xValue  != zValue) && (yValue  != zValue));
model boolean threeEqual;
represents threeEqual <- (xValue == yValue) && (xValue == zValue);

 ensures (!validParameters) && answerString.equals("Invalid  input")
 ||                  (validParameters) && (!nonNegativeValues)  && answerString.equals("Invalid  input")
 ||                  (validParameters) && (nonNegativeValues)  && (!validTriangle)
                              && answerString.equals("Invalid  triangle")
 ||                  (validParameters) && (nonNegativeValues)  && (validTriangle)
                     && (noneEqual) && answerString.equals("Scalene")
 ||                  (validParameters) && (nonNegativeValues)  && (validTriangle)
                     && (!noneEqual) && (threeEqual) && answerString.equals("Equilateral")
                     ||  (validParameters)  && (nonNegativeValues)  && (validTriangle)
                     && (!noneEqual) && (!threeEqual) && answerString.equals("Isosceles");
```

Figure 4 - JML Specification

Integrating JML and Coverage Analysis with the automated testing of Java programs

JMLc generates runtime assertion enabled JVM bytecode in each class file for each of the Java files. One of the outputs from this process is a set of JML-annotated source files augmented with runtime assertion checks. Using the Eclipse IDE, code coverage is performed on the augmented source files, rather than the original source files. Since the augmented source files can be times larger than the original, this is less than useful.

In order to perform code coverage on the original source files, I used Ant (Hatcher, 2007) to control the build, and used Eclipse as an intelligent editor. Two compile/build cycles were used:

- Compile all classes with assertion checks disabled, and coverage enabled. This allows one to determine the number of runs needed to achieve a desired code coverage figure. Further one can use the code coverage tool to inspect an Emma-highlighted version of the source code in order to decide what to do about any sections of the source code which are not covered. Often, the easiest thing to do is to add an extra test to the JUnit test suite. Since the run time of code compiled with javac is shorter than that compiled with jmlc, it's feasible to perform this compile/build cycle with differing test data sizes before using jmlc-compiled classes.
- Compile the uut classes with jmlc, i.e. with assertion checks enabled and coverage disabled. Other test classes will be linked by the JVM. Due to the larger size of the classes, it can take longer to run JML-compiled code than that compiled with javac.

It's useful to take a look at the directory structure used in order to see what's going on:

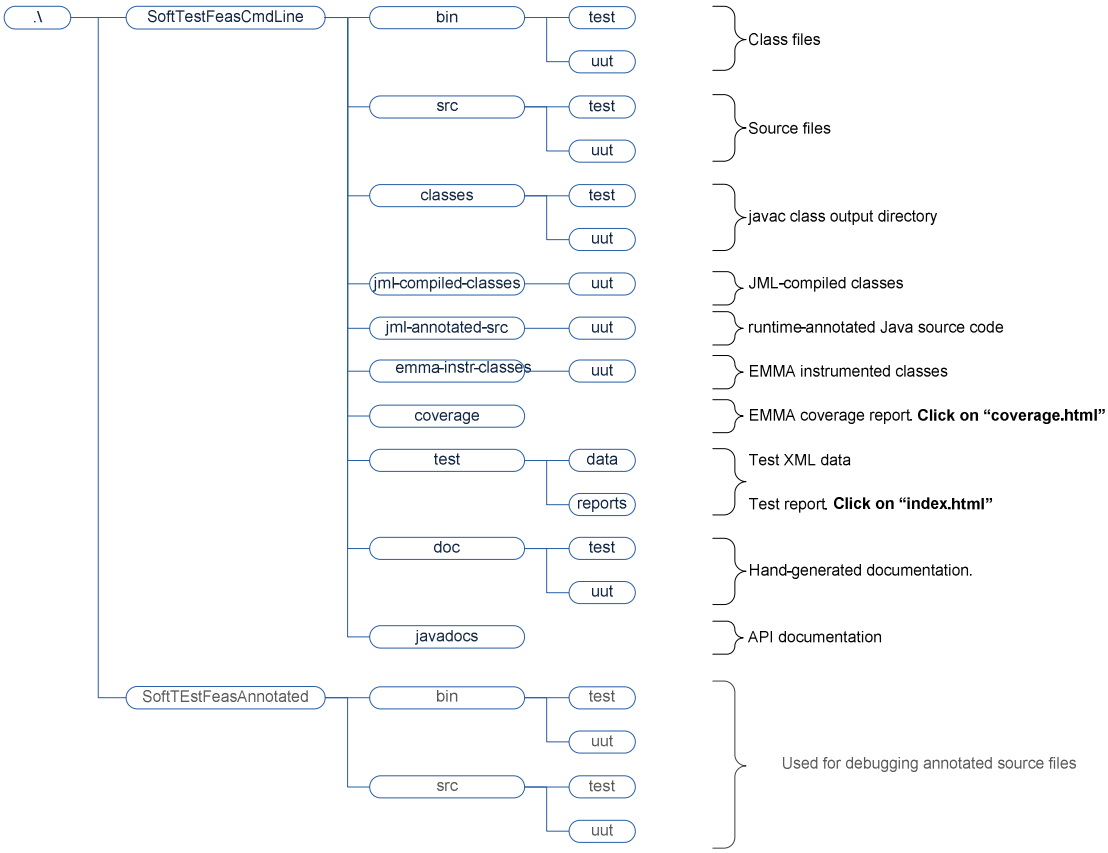

Figure 5 - Directory structure

3. EXPERIMENTAL SETUP

The jmlunit tool generates JUnit test classes that rely on the JML runtime assertion checker (Leavens, G.T. and Cheon, Y, 2006). The test classes send messages to objects of the Java classes under test. When the method under test has an assertion violation, then the implementation failed to meet its specification, and hence the test data detects a failure. In other words, the generated test code serves as a test oracle whose behaviour is derived from the specified behaviour of the class being tested.

UUT Classes

The classes which make up the Unit Under Test (UUT) are shown in Figure 6. The ButtonHandler Inner Class was modified by surrounding the call to showAnswer with a try/catch/finally clause. This latter clears and sets a JMLReady flag in an instance of the ManageJMLErrors class (see below) in order to inform the Test Fixture that showAnswer has finished, and that it is safe to look at the results. It also catches exceptions thrown by showAnswer, and copies them to a JML Error Value variable in the instance of the ManageJMLErrors class. This allows the Test Fixture to examine any exceptions thrown, and to log any corresponding test failures or errors.

Triangle Class

Categorises the triangle based on the side lengths, returning one of: "Invalid triangle", "Equilateral", "Approximately Equilateral", "Isosceles", or "Scalene". Also determine whether triangle is right-angled.

Class Variables

x	stores the length of one of the triangle's sides
y	stores the length of one of the triangle's sides
z	stores the length of one of the triangle's sides
right	whether it's a right-angled-triangle

Tridentify Class

Displays a prompt "Enter triangle side lengths and press identify". Displays three text input fields, a button labelled "Identify" and a blank output text field.

When the user presses the "Identify" button, displays the following output:

- "Invalid input" if any of the inputs are not valid values as per the Java API Specification for a "DoubleValue"
- "Invalid triangle" if the inputs don't form a valid triangle
- "Scalene", "Equaliteral", or "Isosceles" - depending on the triangle

Class Variables

tF_prompt	Text field for user prompt
tF_x	Editable text field for Length 1
tF_y	Editable text field for Length 2
tF_z	Editable text field for Length 3
tF_answer	Text field for result
b_execute	"Identify" command push-button
layout	GridLayout manager that lays out a container's components in a rectangular grid

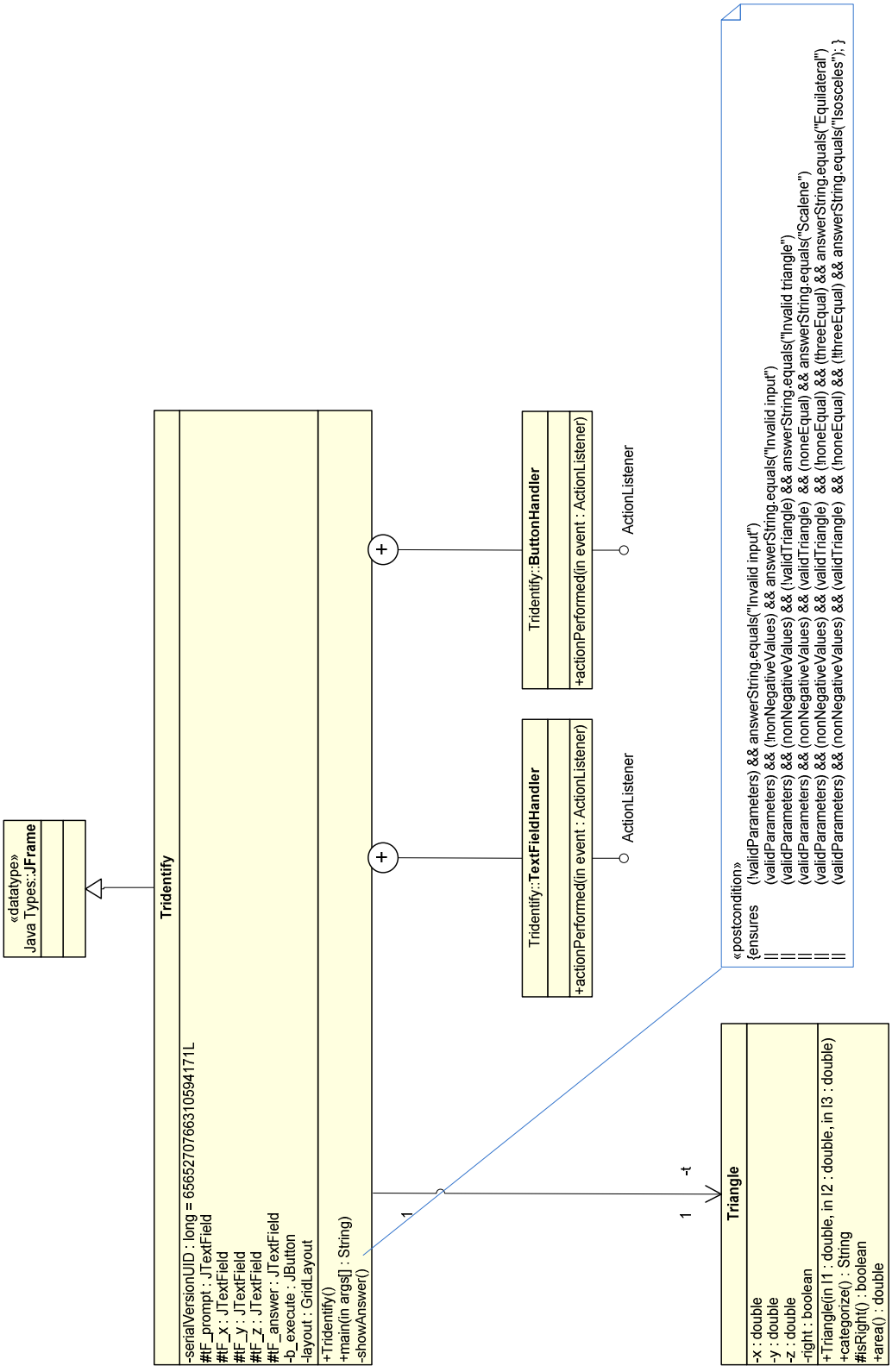

Figure 6 - UUT Classes

Test Classes

The Test Classes comprising the Framework are shown in Figure 7.

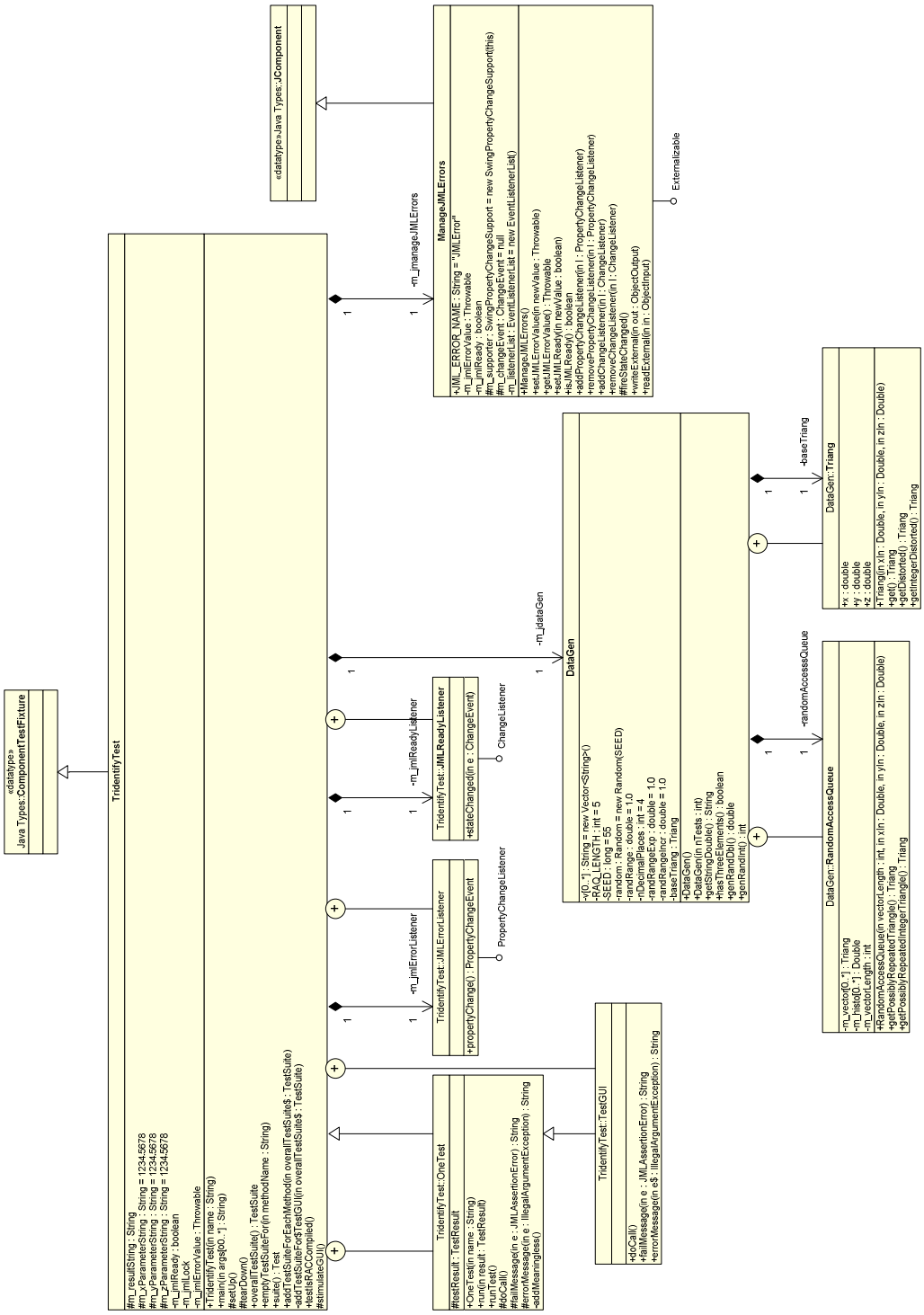

Figure 7 - Test Classes

TridentifyTest Class.

TridentifyTest extends the ComponentTestFixture class, adding some PropertyChangeListener inner classes which are registered with ManageJMLErrors, and a test case inner class for testing the GUI.

A ComponentTestFixture is a subclass of TestCase. It's a Fixture for testing AWT and/or JFC/Swing components under JUnit.

- Ensures proper setup and cleanup for a GUI environment.
- Provides methods for automatically placing a GUI component within a frame and properly handling Window showing/hiding (including modal dialogs).
- Catches exceptions thrown on the event dispatch thread and rethrows them as test failures.

A test case defines the fixture for running multiple tests. We define a test case by:

- Implementing a subclass of TestCase
- defining instance variables that store the state of the fixture
- initialising the fixture state by overriding setUp
- cleaning-up after a test by overriding tearDown.

Each test runs in its own fixture so there can be no side effects among test runs.

Class Variables

m_jmlErrorValue	Used to hold a JMLError resulting from a Property change in an instance of the ManageJMLErrors class A Property change in an instance of the ManageJMLErrors class causes a JML Error to be launched by the Event Dispatch Thread which results in a call to JMLErrorListener.propertyChange(). The New Value of the PropertyChangeEvent parmeter is copied to m_jmlErrorValue
m_jmlReady	producer-consumer flag used to signal that the result of a triangle categorisation is ready. ManageJMLErrors has a version of this flag. The method MLReadyListener.stateChanged() is invoked by ManageJMLErrors in order to force an update of this m_jmlReady, in response to any change in ManageJMLErrors' m_jmlReady
m_jmlLock	m_jmlLock is a synchronisation object used to guarantee atomic access to m_jmlErrorValue and m_jmlReady
m_xParameterString	stores the length of one of the triangle’s sides
m_yParameterString	stores the length of one of the triangle’s sides
m_zParameterString	stores the length of one of the triangle’s sides
m_resultString	stores the result of the classification.

Inner Classes

OneTest A JUnit test object that can run a single test method

The OneTest. runTest() method constructs a tridentify object and runs the test, getting 3-tuples of data from the dataGen Object (See below), repeatedly calling TestGUI.doCall() in order to exercise the GUI.

The doCall() invocation is surrounded by a try/catch block which deals with JMLInternalPreconditionErrors, JMLAssertionErrors and IllegalArgumentExceptions, logging test failures and errors as appropriate.

TestGUI Test GUI, extends OneTest, adding a Producer/Consumer handshaking protocol. If TridentifyTest's copy of the m_jmlErrorValue variable is not empty, its value is thrown as an exception.

JMLErrorListener Property change listener which is registered with ManageJMLErrors (See below). It listens for any change in ManageJMLErrors' copy of m_jmlErrorValue, and updates TridentifyTest's m_jmlErrorValue accordingly

JMLReadyListener Property change listener which is registered with ManageJMLErrors (See below). It listens for any change in ManageJMLErrors' copy of m_ jmlReady, and updates TridentifyTest's m_ jmlReady accordingly

ManageJMLErrors Class

The ManageJMLErrors class manages the transmission of JMLAssertionErrors from Tridentify to TridentifyTest. JMLAssertionErrors are caught by the try-catch block in Tridentify.showAnswers() and are copied to a bound property of m_manageJMLErrors, resulting in a propertyChangeEvent being enqueued in the EDT FIFO. An Event Listener in TridentifyTest will pick up the JMLAssertionError and treat it as either a meaningless test input or else add a failure to the list of failures, as appropriate.

Class Variables

m_jmlErrorValue Used to hold a JMLError,(e.g. a JMLAssertionError) which is a Throwable type. It's a Bound Property of ManageJMLErrors ; whenever m_jmlErrorValue changes, interested listeners are notified via the Event Dispatch Thread.

m_jmlReady Used to hold a producer-consumer flag used to signal that the result of a triangle categorisation is ready. It's a Bound Property of ManageJMLErrors ; whenever m_ jmlReady changes, interested listeners are notified via the Event Dispatch Thread.

Inner Classes

m_listenerList A class that holds a list of EventListeners. A single instance is used to hold all listeners (of all types) for the instance using the list.

DataGen Class

DataGen builds a suite of test data for use by TridentifyTest. Two variants of the set of test data can be generated:

1. A set of hand-generated test data used to validate Tridentify.showAnswer's JML Specification. Some of the test data are derived from the Rules in theStephen's informal Specs for Tridentify. The remainder were derived from Stephen's Tridentify Testing Notes, a document which describes the deliberate errors he inserted into the Triangle class.
2. A set of random test data used to stimulate the Tridentify class. My hope is to find some rules-of thumb (I was going to say 'heuristic', but that would be far too pompous) for generating a set general-purpose randomised test data. One can see if these randomised test data can come close to uncovering a set of faults which is close to the set found by using the hand-generated test data.

   Two sub-sets of random data are generated: the first being divided by 256 in order to provide double data, and the second used to provide integer data. In each case the range of the random number starts out small and increases exponentially as the test progresses. Thus the character of the test data depends heavily on the number of tests used. The numbers generated are used to add 'noise' to the lengths of the sides of a basis triangle to produce triangle objects. These triangles are input into a queue, the data output from the queue are normally taken from it's head, but may be taken from other elements with a probability which decreases as element's index in the queue increases. So, if a particular triangle appears in the test stream, it may be repeated.

Class Variables

v an array of test data comprising 3-tuples: (x,y,z) describing lengths of the sides of triangles.

randomAccessQueue * First-in, random-out (FIRO) queue of Triangles.

Removing an item from the queue removes elements from the queue with decreasing order of probability:

Queue head (highest probability)
position 1 (lower probability)
...
Queue tail (lowest probability)

The algorithm for generating random test data is described above (see Auto-generated test data).

4. RESULTS

Hardware and Software Configuration

Build/Run Environment

Ant is used to build the versions of the program. Ant target dependencies are shown in Figure 8:

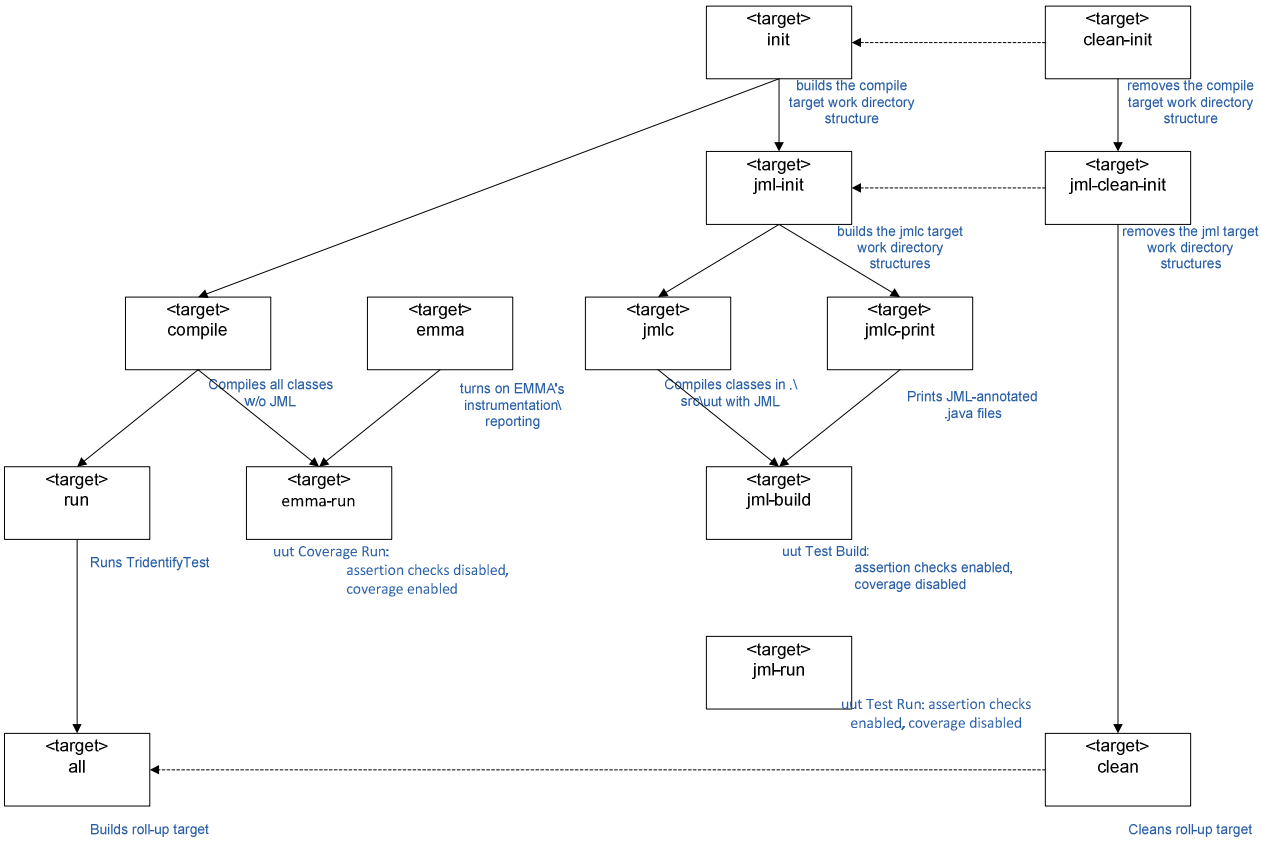

Figure 8 - Ant dependancies

The Eclipse environment is used as an intelligent editor. The following libraries were used:

- C:\junit\junit-4.6-src.jar
- C:\abbot\lib\bsh-2.0b4.jar
- C:\JML\bin\jmljunitruntime.jar
- C:\junit\junit-4.6.jar
- C:\abbot\lib\xml-apis.jar
- C:\JML\bin\jmlmodelsnonrac.jar
- C:\abbot\lib\costello.jar
- C:\JML\bin\jmlmodels.jar
- C:\Apache\commons-validator-1.3.1\commons-validator-1.3.1.jar
- C:\abbot\lib\xercesImpl-2.8.1.jar
- C:\JML\bin\jml-release.jar
- C:\JML\bin\jmlruntime.jar
- C:\abbot\lib\example.jar
- C:\abbot\lib\abbot.jar
- C:\abbot\lib\jdom-1.0.jar
- C:\abbot\lib\junit-3.8.1.jar
- C:\abbot\lib\gnu-regexp-1.1.0.jar
- C:\junit\junit-dep-4.6.jar
- C:\abbot\src.jar

Experimental Results

Code Coverage

I'm reporting results on a code coverage basis here, in order to compare my results with those reported elsewhere (Alzabidi, 2009), (Michael, McGraw and Schatz, 2001).

Hand-Generated Test Data

Running the tests using hand-generated test data resulted in an 89% code coverage for Triangle. The lines not reached were either unreachable or else outside Figure 2's specification:

Line No	Code	Comment
97	`ans = new String("INTERNAL ERROR: unknown type");`	Unreachable
105	`right = (x*x)+(y*y)==(z*z);`	Outside spec
115	`return right;`	Outside spec
124-125	`double s=(x+y+z);` `return Math.sqrt( s*(s-x)*(s-y)*(s-z) );`	Outside spec

28 hand-generated tests were needed to achieve the above coverage.

Randomly-Generated Test Data

Running the tests using randomly-generated test data resulted in an 86% code coverage for Triangle. The lines not reached were:

Line No	Code	Comment
63	`ans = new String("Invalid triangle");`	not feasible to generate a particular special value.
97	`ans = new String("INTERNAL ERROR: unknown type");`	Unreachable
105	`right = (x*x)+(y*y)==(z*z);`	Outside spec
115	`return right;`	Outside spec
124-125	`double s=(x+y+z);` `return Math.sqrt( s*(s-x)*(s-y)*(s-z) );`	Outside spec

32 randomly-generated test objects (16 integer and 16 floating-point) were needed to achieve the above coverage.

The code coverage is the same as for the hand-generated tests, with the following exception:

> We would have to wait a very long time for a pseudo-random sequence to generate a double of value exactly 1234.5678. A quicker option would be to let the tests run for a short time and then inspect of the code coverage for 'special values'. This would motivate the addition of a 'special value' test to the test suite.

Rule Coverage

I'm reporting results on a 'Rule-based" coverage basis here, in order to estimate the "specification coverage"

Hand-Generated Test Data

The hand-generated tests were evolved from:

- the specification for the triangle classification program in Figure 2

```
* More precise version of specification:
*
* 2. When the use presses the "Identify" button, display the output
*    in the output text field using the following rules
*
* Rule  Inputs                              Output
* ----  ----------------------------------  --------------------------
*   1   any of x,y,z are invalid doubles    "Invalid input"
*
*   2   x,y,z all valid text for doubles,   "Invalid triangle"
*       but x,y,z form an invalid triangle
*
*   3   x,y,z all valid text for doubles    "Scalene"
*       and x,y,z form an valid triangle
*       and x!=y!=z
*
*   4   x,y,z all valid text for doubles    "Equilateral"
*       and x,y,z form an valid triangle
*       and x=y=z
*
*   5   x,y,z all valid text for doubles    "Isosceles"
*       and x,y,z form an valid triangle
*       and any two of the sides are equal
```

Figure 2. (Rule 1 through Rule 5)

- 'Testing Notes' (Brown, 2008). (BBT 1 through BBT 4 and WBT 5.2)

Since Rules 1 through 5 are exercised by the first 15 test 3-tuples. The remainder of the tests exercise BBT 1 through BBT 4 and WBT 5.2.

Table 1 - Hand-Generated Test Data

TestID	**Hand-Generated Input Parameters**	**What's being tested**
	(10, 11, 12)	sanity check: Scalene
Rule 1:	(10, 11, Jimmy)	any of x,y,z are invalid doubles "Invalid input"
	(10, Jimmy, 12)	
	(Jimmy, 12, 12)	
Rule 2:	(10, 11, 22)	x,y,z all valid text for doubles, "Invalid triangle" but x,y,z form an invalid triangle
Rule 3:	(10, 11, 12)	x,y,z all valid text for doubles "Scalene" and x,y,z form an valid triangle and x!=y!=z
	(11, 12, 10)	

	(12, 10, 11)	
Rule 4:	(10, 10, 10)	x,y,z all valid text for doubles "Equilateral" and x,y,z form an valid triangle and x=y=z
Rule 5:	(12, 10, 10)	x,y,z all valid text for doubles "Isosceles" and x,y,z form an valid triangle and any two of the sides are equal
	(10, 12, 10)	
	(10, 10, 12)	
	(2, 10, 10)	
	(10, 2, 10)	
	(10, 10, 2)	
BBT: 1.	(NaN, NaN, NaN)	Try "NaN" and "Infinity" as input values - they work (spec error: should ref FloatingPointLiteral not FloatValue)
	(POSITIVE_INFINITY, POSITIVE_INFINITY, POSITIVE_INFINITY)	
BBT: 2.	(10, 10, NaN)	Invalid "z" will raise an unhandled exception - implementation error
BBT: 3.	(-10, -10, -10)	negative values of x,y,z are categorised as triangles - error of omission
	(-10, 10, 10)	
	(10, -10, 10)	
	(10, 10, -10)	
BBT: 4.	(12, 10, 11)	If x>y for a scalene triangle, returns isosceles
	(1234.5678, 1234.5678, 1234.5678)	
	(1234.5678, 10, 10)	
	(10, 1234.5678, 10)	
	(10, 10, 1234.5678)	
WBT: 5.2	(10.001, 10, 10)	If x, y, and z are approx equal (within 0.1%) get "Approximately Equilateral"

The test results using the above hand-generated test data were correct. This validated the JML specifications that I wrote.

Randomly-Generated Test Data

The randomly-generated input parameters (recall that these parameters are constrained so that each set describs a variant of a basis object), exercise Rules 2 through 5 and BBT 3, 4 and WBT 5.2. Rule 1 and BBT 1 and 2 require inputting parameters that can't be generated by a random number generator. If needed, these cases can be added to the set of JUnit cases.

Table 2 shows the set of randomly-generated variants of the basis triangle used to test the classifier, (via its GUI interface), the resultant category assigned to each object and the corresponding TestID from Table 1.

Table 2 - Randomly-Generated Test Data

Randomly-Generated Input Parameters	Category Output	TestID
(1.0, 1.0, 1.0)	Equilateral	Rule 4
(0.998, 0.9985, 0.9972)	Approximately Equilateral	WBT 5.2
(0.9916, 1.0014, 0.987)	Scalene	Rule 3
(0.9837, 0.9796, 0.9823)	Isosceles	BBT 4
(1.019, 0.9836, 1.0323)	Isosceles	BBT 4
(1.0354, 1.0589, 1.0198)	Scalene	Rule 3
(0.8859, 0.8983, 0.9186)	Scalene	Rule 3
(0.8859, 0.8983, 0.9186)	Scalene	Rule 3
(1.0059, 0.41990000000000005, 1.3087)	Isosceles	BBT 4
(0.4769, 0.7008, 1.3947)	Invalid triangle	Rule 2
(0.7625, -1.2172999999999998, -1.7706)	Isosceles	BBT 3, 4
(2.4377, -2.3895, 2.1061)	Invalid triangle	BBT 3,
(-4.2989, 6.0142, 9.3727)	Invalid triangle	BBT 3,
(2.0982000000000003, -6.1007, -2.108)	Isosceles	BBT 3, 4
(0.7625, -1.2172999999999998, -1.7706)	Isosceles	BBT 3, 4
(2.4377, -2.3895, 2.1061)	Invalid triangle	BBT 3, Rule 2
(1.0, 1.0, 2.0)	Isosceles	Rule 5
(3.0, 4.0, 5.0)	Scalene	Rule 3
(5.0, 10.0, 7.0)	Scalene	Rule 3
(2.0, 3.0, 21.0)	Invalid triangle	Rule 2
(3.0, 4.0, 5.0)	Scalene	Rule 3
(54.0, 44.0, 24.0)	Isosceles	BBT 4
(10.0, 121.0, 127.0)	Scalene	Rule 3
(172.0, 158.0, 27.0)	Isosceles	BBT 4
(76.0, 327.0, 726.0)	Invalid triangle	Rule 2
(10.0, 121.0, 127.0)	Scalene	Rule 3
(10.0, 121.0, 127.0)	Scalene	Rule 3
(1519.0, 2982.0, 1794.0)	Scalene	Rule 3
(3378.0, 6892.0, 4070.0)	Scalene	Rule 3
(3378.0, 6892.0, 4070.0)	Scalene	Rule 3
(13781.0, 2620.0, 11933.0)	Isosceles	BBT 4
(18270.0, 14243.0, 992.0)	Invalid triangle	Rule 2

With the exception of Rule 1 and of BBTs 1 and 2, a short sequence of randomly altered basis objects worked surprisingly well, covering all the JML specifications.

The idea of compiling the uut classes with assertion checks disabled, in order to ascertain the coverage, and following this by adding extra tests to the JUnit test suite to ensure coverage before compiling the classes with assertion checks enabled and coverage disabled, seems to be a highly effective technique.

Relative difficulty of using Hand-generated test data vs. using auto-generated test data

Once JML model variables are in place, bridging the gap between low-level details and a high level of abstraction, it's not difficult to write a concise JML *ensures* clause to specify the behaviour of the GUI-based application, given a half-decent informal specification to begin with.

For the case of the triangle classifier, hand-generating test data to exercise Rules 1 through 5 isn't difficult, and, hand-generating test data to exercise the BBTs and WBTs isn't much harder. However, figuring out what to test in these cases is a task which needs to be done by a very experienced test engineer.

It is this latter difficulty which makes the use of auto-generated test data so attractive.

5. CONCLUSIONS AND FUTURE WORK

I've shown that JML specifications combined with run-time assertion checking can act as a Test Oracle for GUI programs. I've also shown that automatically generating a set of randomised Objects abstracted from the GUI's specification makes for a particularly efficient set of test cases, in the case of the triangle classification program. I've demonstrated how runtime assertion checking and coverage analysis can be integrated with automated testing of GUI code.

The idea of producing as test data, a set of randomly distorted test objects rather than a set of random numbers, seems to work well, especially starting as we do with a 'perfect' object and then distorting this more and more as the test progresses, and in addition, the idea of generating half the random data as integers, and half as doubles, seems to have a tendency to hit corner cases.

In the case of the Tridentify/Triangle classes, good results were found after generating only 32 test objects. It would be interesting to see how well this model of test object generation works other Classes, and whether or not a set of heuristics for generating randomly distorted test objects of general applicability can be evolved.

The above approach means that the test engineer can spend his time writing JML GUI specifications at a high level of abstraction, rather than hand-generating test cases, if the idea of automatically generating sets of randomised test objects does turn out to be generally applicable.

REFERENCES

Alzabidi Kumar, and Shaligram Automatic Software Structural Testing by Using Evolutionary Algorithms for Test Data Generations [Article] // International Journal of Computer Science and Network Security. - 2009. - 4 : Vol. 9.

Brown Stephen Tridentify Testing Notes // Private communication. - Maynooth : National University of Ireland, Maynooth, 2008.

Cai Li and Ning Optimal software testing in the setting of controlled Markov chains [Article] // European Journal of Operational Research. - Maryland Heights : Elsevier, 2005. - 2 : Vol. 162. - pp. 522-579.

Chan Towey, Chen, Kuo and Merkel Using the Information: Incorporating Positive Feedback Information into the Testing Process [Conference] // Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice. - Washington : IEEE Computer Society, 2003. - pp. 71 - 76.

Chen, Shen, and Chang GUI Test Script Organization with Component Abstraction [Conference] // Proceedings of the 2008 Second International Conference on Secure System Integration and Reliability Improvement. - Washington : IEEE Computer Society, 2008. - pp. 128-134.

Hamlet Richard Random testing [Book Section] // Encyclopedia of Software Engineering / ed. Marciniak J.. - New York : Wiley, 1994.

Hatcher E. and Loughran, S. Ant in Action [Book]. - Greenwich : Manning, 2007.

Hu Jiang and Cai Adaptive Software Testing in the Context of an Improved Controlled Markov Chain Model [Conference]. - Washington : IEEE Computer Society, 2008. - pp. 853-858.

Jones Sun and Specification-Driven Automated Testing of GUI-Based Java Programs [Conference] // Proceedings of the 42nd annual Southeast regional conference. - New York : ACM, 2004. - pp. 140 - 145.

Leavens G. T. and Cheon, Y A Simple and Practical Approach to Unit Testing: The JML and JUnit Way [Article] // Lecture Notes in Computer Science, Springer-Verlag, 2002, pages 231-255. - Malaga : [s.n.], 2002. - Vol. 2374.

Leavens, G.T. and Cheon, Y Design by Contract with JML [Online]. - Iowa State University, December 2006. - 26 01 2010. - ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf.

Michael, McGraw and Schatz Generating Software Test Data by Evolution [Article] // IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. - Piscataway : IEEE Press, 2001. - 12 : Vol. 27. - pp. 1085 - 1110.

Murphy Christian, Kaiser Gail, and Arias Marta Parameterizing Random Test Data According to Equivalence Classes [Conference] // Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007). - New York : ACM, 2007. - pp. 38 - 41.

Myers G.J. The Art of Software Testing [Book]. - New York : John Wiley, 2004.

Poll Breunesse Verifying JML specifications with model fields [Conference] // Formal Techniques for Java-like Programs. Proceedings of the ECOOP'2003 Workshop,. - Zürich : ETH Zürich, 2003. - pp. 51–60.

Roubtsov V EMMA User Guide [Online]. - sourceforge.net, 2006. - 26 01 2010. - http://emma.sourceforge.net/userguide_single/userguide.html.

Schneider A. JUnit best practices [Online]. - javaWorld, 2000. - 26 01 2010. - http://www.javaworld.com/jw-12-2000/jw-1221-junit.html.

Sebern M. SE-3811 Tool Installation [Online] // SE-3811 Formal Methods. - EECS Department, Milwaukee School of Engineering, 2008-2009. - 26 01 2010. - http://people.msoe.edu/sebern/courses/se3811/tools/index.shtml.

Verzulli J. Getting started with JML: Improve your Java programs with JML annotation [Online]. - IBM developerWorks, March 2003. - 26 01 2010. - http://www.ibm.com/developerworks/java/library/j-jml.html.

Wall Timothy Getting Started with the Abbot Java GUI Test Framework [Online] // Abbot Java GUI Test Framework. - sourceforge.net, 2008. - 17 01 2010. - http://abbot.sourceforge.net/doc/overview.shtml.