

ON AUTOMATED CHECKING OF JAVA APPLETS

Scott Dexter
Department of Computer and Information Science
Brooklyn College of CUNY
Brooklyn, NY 11210
(718) 951-3125
sdexter@sci.brooklyn.cuny.edu

ABSTRACT

Through an initiative geared to create "partially virtual" curricula for the courses that comprise Brooklyn College's Core Curriculum, we are developing a "virtual" module to introduce programming to non-majors in the context of Java applets. An integral part of this effort is a mechanism to provide instantaneous automated feedback on student work. We illustrate techniques, based primarily on Java's support for reflection, for providing automated checking of and feedback on Java applet programming exercises.

1 INTRODUCTION

The Web's interactive nature offers enormous possibilities for innovative pedagogy (see e.g. [1,2,3,4]); in computer science, this interactivity provides an opportunity for us to offer beginning programmers focused and immediate automated responses to their work. Simultaneously, the rapid development of the Web and the software that undergirds and enriches it has placed sophisticated GUI applications within the reach of these beginning programmers. Indeed, some argue that introducing programming using "traditional" text-based models in an increasingly graphically oriented environment is both overly restrictive and pedagogically inadequate [5]. Getting the best of both worlds, however — that is, teaching GUI programming *and* using the Web to provide automated responses — is not a straightforward proposition: it is not immediately obvious how to automate the testing of a GUI application, since these generally rely on a user's interaction with the graphical representation of the GUI components.

This paper treats the specific case of providing automated feedback for Java applets. We will show that by exploiting unique features of Java, particularly *reflection*, we can provide automated and highly focused feedback. Moreover, these tests offer an important pedagogical benefit: in the course of creating the tests, the instructor must consider all aspects (and hence all potential pitfalls) of creating the applet. Thus, in addition to providing feedback regarding the applet's behavior (e.g. "does the right thing when the user clicks that button?"), is forced to consider, test for, and contemplate providing feedback for errors that may arise at all stages of development.

2 BACKGROUND

Brooklyn College recently received a grant from FIPSE to evaluate the effectiveness of asynchronous (or “distance”) learning in the context of the –Brooklyn College Core Curriculum. Instructors in all areas of the Core are developing “partially virtual” versions of the Core courses: that is, versions of the course in which one-third to one-half of the class sessions are replaced by on-line activities (e.g. Web-mediated discussion, “virtual laboratories,” etc.). These will then be offered over the next several semesters in parallel with “traditional” sections of the Core in order to provide the foundation for a rigorous evaluation of the effectiveness (along several axes) of asynchronous learning.

The Computer and Information Science department participates in the Core through *Introduction to Mathematical Reasoning and Computer Science*, administered jointly with the Mathematics department. The course is intended to address fundamental issues of both mathematics and computer science. Typically, this is done by introducing concepts from a handful of areas of mathematics (e.g. logic, probability, and/or number theory), then discussing the history and development of computers and computation and providing a gentle introduction to programming in some language.

A group of faculty from the CIS department has engaged the challenge of creating a partially virtual version of the course, with the particular goal of creating an asynchronous learning module for introducing programming to non-majors. This choice was motivated in part by the presence of WebToTeach, a Web-based environment for managing programming exercises that was developed at Brooklyn College and is currently being used in a variety of courses for majors [6]. Motivated by the observations that

- Large sets of exercises reinforcing “micro-concepts” are an integral (but not exclusive) component of mathematics and foreign language pedagogy — which are both closely related to programming, and
- The multiplicity of both right and wrong answers for even small-scale programming exercises throws up a significant logistical obstacle to providing timely and effective evaluation and feedback,

WebToTeach provides an environment in which

- instructors can develop exercises (by providing both instructions for the student and a set of automated tests to be applied to student submissions),
- students can develop and submit their solutions and receive instantaneous feedback based on the results of the instructor-specified tests, and
- instructors can monitor the progress of individuals and classes and offer suggestions as students develop their solutions.

Once we decided to use WebToTeach, it remained to choose a pedagogical framework for its use. Because this course is required of *all* students (not just, say, intended science majors), providing a credible motivation for programming is particularly challenging. We chose to leverage the allure of the Web and teach programming in the context of Java applets. Among the particular advantages of this approach are

- Applets are “real” programming, as can be illustrated in the “introduction to computing (and the Internet)” component of the course;
- Applets produce visually appealing output.

These advantages may potentially be weakened by the structural (and syntactic) complexity of applets, but we anticipate that the gradual approach that the “micro-exercise” emphasis of WebToTeach encourages will allow us to present applets in a friendly fashion.

3 FEEDBACK AND ASYNCHRONOUS LEARNING

Our experience using WebToTeach simply as a “homework management” tool in non-virtual classes has shown — not surprisingly — that providing immediate feedback about a submission substantially increases the proportion of students who successfully complete the homework. That is, when students have (limited) access to the results of a battery of tests, they are likely to return to the problem and try to discover the nature of the flaws in their submissions, rather than applying a handful of (usually insufficient) tests of their own design, turning it in, and receiving both the bulk of their feedback and an evaluation at the same time.

In the context of asynchronous learning, useful and timely feedback takes on a new kind of importance. No longer simply an aid to turning in high-quality homework, it is of primary significance as students reckon with the course content and gauge their own understanding of the material. Thus, the feedback received by the students must do more than alert them to problems with their submissions: it must also function in dialog with the student, providing meaningful guidance and assisting in the refinement of both conceptual and pragmatic understanding.

3.1 Feedback vs. Graphical Output

Our decision to teach applets, with their strong graphical orientation, might seem to render moot most discussion of how to provide appropriate feedback. Certainly, in the case of a simple introductory exercise, such as

Write an applet that draws a circle inside a rectangle.

the output of the applet would seem to provide all necessary feedback: either there is a circle inside a rectangle, or there is not. Based on this type of feedback, the student should be able to return to the source code and refine it.

But while this feedback might indeed suffice for an intermediate programmer, it is clearly insufficient for the rank novice — especially one who might not be strongly motivated. In this case, guidance is required well before the applet is successfully executed, both to provide context-specific interpretation of compiler complaints and to take advantage of student’s errors to highlight and reinforce key concepts.

3.2 Providing Feedback on Graphical Output

But the blessing of graphical output — that (in most cases) its correctness can be assessed easily and quickly by the student — is also a curse, in that automated analysis of the output ranges, in general, from difficult to impossible. Whereas it is exceedingly easy both to customize the input and to analyze structurally the output of a text-based program, the output of an applet is not easily susceptible to capture. Of course, some

errors may be found via a static analysis of the code — and for simple exercises, this might even suffice to identify a majority of correct solutions. But in general, it is clear that only by executing the applet can we hope to acquire enough information about it to provide meaningful feedback, both positive and negative.

The key to the effective resolution of this problem is the utilization of Java’s *reflection* capability — that is, the ability of a Java program to inspect and manipulate itself. We will see that this tool allows an arbitrary level of feedback for a variety of potential problems.

4 REFLECTION IN JAVA

The introduction of Java 1.1 added to Java the capacity for programs to analyze and manipulate Java classes (including classes that are *not* part of the program itself). This capacity is of clear utility to various types of browsers, parsers, and builders, and is a crucial part of the JavaBeans framework (in which a “bean” is an object that may be manipulated and customized using a visually oriented tool). In general, reflection is likely to be useful in any program that deals with classes dynamically. [7]

Packaged as the Reflection API, this tool rests on the ability of Java programs to use class files (i.e. the compiled source code) both as code *and* as data. Classes, constructors, fields, members, and methods may all be represented as data objects with appropriate methods. For example, the `newInstance()` method of the `Constructor` class returns a new instance of the associated constructor’s declaring class; the `invoke()` method of the `Method` class invokes the associated method; the `get()` and `set()` methods of the `Field` class manipulate the value of the associated field.

Checking and testing GUI code relies heavily on the dynamic manipulation of the classes involved, as we will show below. Using reflection to apply tests to student submissions allows us to detect and analyze a wide variety of runtime and semantic errors; in tandem with WebToTeach’s facility for providing customized error messages at compile time, this allows us to provide a full spectrum of specific feedback. We will see that the testing process itself, in this context, must by nature of its structure mirror the development process, thereby ensuring that the instructor has every opportunity to exploit the advantages offered by reflection.

5 EXAMPLE: TESTING A SIMPLE APPLLET

In this section, we will highlight some of the important features of using reflection to test applets. We will consider the following simple exercise:

Write a complete Java applet called `OvalAndRectApplet` that draws a (hollow) rectangle and a (hollow) oval anywhere on the screen so that they do not overlap.

Certainly, this is easily solved; one possible solution is

```
import java.awt.*;
import java.applet.*;

public class OvalAndRectApplet extends Applet {
    public void paint (Graphics g) {
        g.drawRect(100,100,50,200);
        g.drawOval(150,300,10,10);
    }
}
```

But in terms of runtime testing, there are a number of possible problems for which we must test, in particular

- declaration of `OvalAndRectApplet` (outside the context of `WebToTeach` this would be checked at compile time via comparison with the filename, but `WebToTeach` hides the filesystem from the user)
- declaration of `paint()` method
- correct parameters of `paint()` method
- fulfillment of specifications

Each of these can be checked in a test driver using reflection, as we show below.

5.1 Correct Declaration of Applet Class

`WebToTeach` allows us to define tests in a variety of ways; in this context it is most appropriate for us to define a test driver class which contains the `main()` method. The first thing this method must do is check to see whether `OvalAndRectApplet` was declared correctly:

```
public class Driver {
    public static void main(String args[]) {
        Class oara = null;
        Method p = null;

        try {oara = Class.forName("OvalAndRectApplet"); }
        catch (ClassNotFoundException e) {
            System.out.println("It looks like
                               OvalAndRectApplet was not defined
                               correctly.");
            System.exit(1);
        }
    }
}
```

Here we attempt to find a class with the name “`OvalAndRectApplet`,” if successful, we save a reference to its `Class` object, otherwise, we print a message (which could of course be more detailed than this one) and exit.

5.2 Correct Declaration of `paint()` Method

Having established that the desired applet class does exist, we must next determine whether the `paint()` method was declared correctly. Since Java permits method overloading, we must check not for an arbitrary `paint()` method, but for one specifically declared to accept one parameter of type `Graphics`:

```
Class[] paramtypes = {Class.forName("Graphics")};

try { p = oara.getMethod("paint", paramtypes); }
catch (NoSuchMethodException e) {
    System.out.println("It looks like you did not
                       provide 'paint.'");
    System.exit(3);
}
```

```
}
```

We first construct an array listing the parameter types, then attempt to extract the `paint()` method (with those parameters) that is defined in the `OvalAndRectApplet` Class. (As before, the output could easily be augmented with suggestions for solving the problem.)

5.3 Fulfillment of Specifications

If the `paint()` method is declared correctly, then all that is left is to assess whether the `paint()` method does what is expected. Performing this assessment requires us to exploit another Java feature: package-based namespaces. In this example, the correct behavior of the `paint()` method simply amounts to correct utilization of the `Graphics` parameter. But in this context it is not appropriate for us to use the “standard” `Graphics` class (i.e. the one defined in the `java.awt` package) — because

1. the behavior of that class involves drawing on the display which, as we noted, is effectively unanalyzable, and
2. the `Graphics` parameter itself is intended to be provided automatically by the applet framework, which we intentionally circumvent during testing.

The solution is for us to define our own `Graphics` class, which will reside in the “default” package. Since the compiler consults this package first, all references to `Graphics` in the applet or the test driver will be to *this* `Graphics` class rather than `java.awt.Graphics`. We can define¹ this class to have just three methods: `drawRect()` and `drawOval()`, which will simply record their parameters, and `isGood()`, which analyzes the values of the parameters to determine whether the specifications were fulfilled (and may also print messages if the specifications are *not* fulfilled). Thus, the remainder of the test driver is:

```
Graphics g = new Graphics();
Object[] params = {g};
OvalAndRectApplet oaraObj = new OvalAndRectApplet();

/* these technically required; never happen */

try { p.invoke(oaraObj,params); }
catch (IllegalAccessException e) {
    System.exit(2);
}
catch (InvocationTargetException e) {
    System.exit(2);
}

if (g.isGood())
    System.exit(0);
else {
    System.out.println("Be sure the shapes do not
                        overlap!");
    System.exit(5);
}
```

¹ Space constraints prevent us from giving the (straightforward) class definition here.

```
}
```

We must create a `Graphics` object to be passed to the `paint()` method; we must also create an applet object whose `paint()` method we will invoke (until this point we have only dealt with a `Class` object which represented the `OvalAndRectApplet` class; we have not declared an actual instance of the `OvalAndRectApplet` class). Then we invoke the `paint()` method (represented by the `Method` object `p`) of object `oaraObj`, with parameter list containing only `g`. When `paint()` completes, we may then invoke `g.isGood()` to analyze the applet's behavior.

6 EXAMPLE: TESTING GUI-BASED APPLETS

The example above, while adequate for illustrating a basic application of the Reflection API, does not address the larger issue of testing more sophisticated applets that use Java's GUI features, incorporating window components (e.g. buttons, textfields, etc.) and the Java event model. In this section we highlight the challenges that lie herein and offer some solutions based, again, on reflection. We will consider testing a solution for the following exercise²:

Write an applet called `PlusAndMinus` which uses two textfields, two buttons (labeled "Add" and "Subtract") and one label. When the "Add" button is clicked, the sum of the values in the text fields is placed in the label; when the "Subtract" button is clicked, the absolute value of the difference of the two values is placed in the label.

To implement a correct solution, the student must provide

- the correct number and type of components (2 Buttons with correct labels, etc.),
- a correct definition of `init()` method (to add components to display), and
- correct interaction among components

In our discussion, we will focus primarily on testing the interaction among components (clearly the most interesting part), but it should become clear how the other checks might be administered, either in a separate piece of code or in tandem with the testing of component behavior. Further, it will be clear that reflection techniques force the instructor to examine all these aspects of the applet's design.

6.1 Invocation of `init()`

Just as the previous example was primarily concerned with the definition and invocation of the `paint()` method, in the context of GUI applet programming we must first address the invocation of `init()`, since this is the method that creates the GUI components and places them on the display.

```
public class Driver {
    public static void main(String args[]) throws
        NoSuchElementException, IllegalAccessException,
        ClassNotFoundException {

        Class pam = null;
        Method i = null;
```

² Appendix A contains a sample solution for this exercise.

```

try { pam = Class.forName("PlusAndMinus"); }
catch (ClassNotFoundException e) {
    System.out.println("Looks like PlusAndMinus
                        was not declared correctly.");
    System.exit(1);
}

Class[] paramtypes = {};

try { i = pam.getMethod("init",paramtypes); }
catch (NoSuchMethodException e) {
    System.out.println("It looks like you did not
                        provide 'init.'");
    System.exit(3);
}

Object[] params = {};
PlusAndMinus pamobj = new PlusandMinus();
try { i.invoke(pamobj,params); }
catch (IllegalAccessException e) {
    System.exit(4);
}
catch (InvocationTargetException e) {
    System.exit(4);
}

```

6.2 Accessing Fields

Testing the behavior of the applet requires the ability to

- put values in the TextFields
- simulate a click on one of the Buttons
- examine the resulting value in the Label

When the applet is actually running, these tasks are easy enough to perform, but when we are testing, without the benefit of a real graphics environment, we must expend a little more effort. The first hurdle is simply that of accessing the fields of the applet, which should be declared private. The use of reflection does not *a priori* revoke access restrictions; in general, a private field is as difficult to access using reflection as it is to access directly. However, it is possible in most cases to override access control.³ The class `AccessibleObject`, which is the superclass of the `Field`, `Method`, and `Constructor` classes, defines a group of `setAccessible()` methods which allow us to override the private visibility of the fields in our applet:

```

Field[] flist = pam.getDeclaredFields();
AccessibleObject.setAccessible(flist,true);

```

First we get an array of the `Fields` declared in the `PlusAndMinus` class, then allow them all to be accessed later on.⁴

³ The ability to override access control is moderated by Java's `SecurityManager` class.

⁴ We note that the applet framework itself provides another technique for accessing components: the `getComponents()` method defined in one of Applet's superclasses. This provides an array of `Components` (the superclass of `Button`, `TextField`, etc.) which may manipulated using

Once we are allowed access to the fields (or, more precisely, to their representations as instances of `Field`), we must then identify those fields that correspond to `TextFields`, those that are `Buttons`, and the `Label`. To do this, we use the `getType()` method of the `Field` class, which returns a `Class` object representing the type of the field:

```
Field[] tflist = new Field[2];
Field[] blist = new Field[2];
int tfCount = 0;
int bCount = 0;
Field reslab = null;
for (int j=0; j < flist.length; j++) {
    if (flist[j].getType() ==
        Class.forName("java.awt.TextField"))
        tflist[tfCount++] = flist[j];
    if (flist[j].getType() ==
        Class.forName("java.awt.Button"))
        blist[bCount++] = flist[j];
    if (flist[j].getType() ==
        Class.forName("java.awt.Label"))
        reslab = flist[j];
}
```

We note that in this code fragment we are assuming the number of each type of component is exactly as we expect. Justifying this assumption mandates that we also test for the type and number of components before we test for correct behavior; again, we see that any testing of the applet necessitates a *thorough* battery of tests.

6.3 Testing Behavior

Now that we have access to the fields, we can use them to test the behavior of the applet. We will consider a simple test that provides one check on the Add button's functionality; in practice this single test will be augmented by a more complete suite of tests (and most of the testing procedure described below would be encapsulated into a method providing easy augmentation and modification of the test suite).

First, we must load some values into the `TextFields`, which may most succinctly be done this way:

```
((TextField) tflist[0].get(pamobj)).setText("10");
((TextField) tflist[1].get(pamobj)).setText("15");
```

The `get()` method of the `Field` class returns the actual value of the field in the given instance; since this method always returns `Object` values, we must cast the result into `TextField` before calling the method (`setText()`) to load a value into the `TextField`.

Then we must determine which of the `Button Fields` is the one labeled "Add" (we check for the label since this is how the problem was specified):

```
Button addBtn = new Button();

int j=0;
```

techniques similar to those discussed here. However, we believe the universality of reflection allows a tighter conceptual framework and gives rise to more general techniques.

```
while (addBtn.getLabel() != "Add")
    addBtn= (Button) blist[j++].get(pamobj);
```

Again, we use `Field.get()` to access the `Button` instances declared in `pamobj`, then invoke the `Button.getLabel()` method to examine the `Button`'s label.

Once we have the "Add" `Button`, we can find out what happens when it is clicked. This requires us to create an event that simulates the user clicking on the button:

```
ActionEvent test =
    new ActionEvent(addBtn,
                    ActionEvent.ACTION_PERFORMED, "Add");
pamobj.actionPerformed(test);
```

The new `ActionEvent` (this is the class of events which represents any button click, as well as a number of other component events) appears to originate from `addBtn` and is of type `ACTION_PERFORMED`; the third field (the "command") is required but not necessary in this situation. We then pass this event to the applet's event handler.

Last but not least, we must confirm that the `Label` value was set correctly:

```
Label res = (Label) reslab.get(pamobj);
if (res.getText() == "25")
    System.exit(0);
else {
    System.out.println("The Add button does not behave
        correctly.");
    System.exit(1);
}
}
```

We simply extract the `Label` value of our `PlusAndMinus` object, then test its contents.

7 CONCLUSION

Java's Reflection API provides an enormous opportunity for providing automated feedback to beginning programmers. The very process of designing reflection-based tests, in fact, concretizes the steps the student must go through in order to produce a correctly functioning applet (or application), and ensures that the instructor considers appropriate feedback for every step (or mis-step) of the development process. While the examples we have discussed here are by necessity relatively simple, the techniques we introduce are clearly generalizable to a wide variety of exercises — limited only by the instructors willingness to cope with the complexity of providing automated testing for increasingly complex programs.

REFERENCES

- [1] Barker, S. "CHARLIE: A Computer-Managed Homework, Assignment, Response, Learning and Instruction Environment," *FIE '97*.
- [2] Medley, D. M. "On-line finals for CS1 and CS2," *ITiCSE '98*.

- [3] Price, R., Ruehr, F., & Salter, R. "Web-based Laboratories in the Introductory Curriculum Enhance Formal Methods," *Proceedings of the 27th SIGSCE Technical Symposium, SIGSCE Bulletin 28, 1*, March, 1996.
- [4] Naccache, H., Lindquist, T., & Urban, S. "A Framework and Reusable Tools for Developing Interactive Course Web Sites," *FIE '98*.
- [5] Skolnick, Michael M. & Spooner, David L. "Graphical User Interface Programming in Introductory Computer Science," *NECC '95*.
- [6] D. Arnow & O. Barshary. "WebToTeach: An Interactive Focused Programming Exercise System," *FIE '99*.
- [7] Horstmann, Cay S. & Cornell, Gary. *Core Java 1.2, Volume 1 — Fundamentals*. Sun Microsystems, Inc. 1999.

APPENDIX A: SAMPLE SOLUTION, PLUSANDMINUS APPLLET EXERCISE

```
public class PlusAndMinus extends Applet
    implements ActionListener {
    public void init() {
        tf1= new TextField(10);
        tf2= new TextField(10);
        result= new Label("      ");
        add = new Button("Add");
        subtract = new Button("Subtract");
        this.add(add);
        this.add(subtract);
        this.add(tf1);
        this.add(tf2);
        this.add(result);
        add.addActionListener(this);
        subtract.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        String s1 = tf1.getText();
        int k1 = Integer.parseInt(s1);
        String s2 = tf2.getText();
        int k2 = Integer.parseInt(s2);
        int ans=0;

        if (ae.getSource() == add)
            ans = k1 + k2;
        else if (ae.getSource() == subtract)
            ans = Math.abs(k1 - k2);
        result.setText(""+ans);
    }

    private TextField tf1, tf2;
    private Label result;
    private Button add,subtract;
}
```