SIMPLE PROBLEM SOLVING IN JAVA:

A PROBLEM SET FRAMEWORK

Viera K. Proulx. Richard Rasala, Jason Jay Rodrigues
College of Computer Science
Northeastern University
Boston, MA 02115
617-373-2462
vkp@ccs.neu.edu, rasala@ccs.neu.edu, jjayr@ccs.neu.edu

ABSTRACT

We present an application that allows for easy creation of simple problem solving exercises in Java, providing robust and safe I/O as well as a basic graphics window. We discuss possible uses for unit testing of classes and explore how the design of this application can be a case study in an object oriented design course.

1. INTRODUCTION

Java is becoming the programming language for introductory courses. Looking through a plethora of textbooks, it is clear that one of the hardest problems is to provide students with an environment where simple program segments can be easily tested with minimal overhead. Some of the solutions such as the GUI package by Koffman and Wolz [5] and breezyGUI by Lambert and Osborne[6] try to solve the problem by creating very elementary graphical user interfaces. Others such as BlueJ from Monash University [4] and MiniJava from Stanford [9] provide entire separate environments for this purpose.

In this paper we describe our solution that is based on the Java Power Tools [7,8]. With the Java Power Tools, it is possible to rapidly create sophisticated graphical user interfaces for complex programs. However, that is *not* what we wish to discuss here. For simple problems, *we want the interface to be created entirely automatically with no manual intervention by the faculty member or the student*. We will explain how we create an action button for each member function of the "problem set class" that is **public**, **synchronized**, **void**, *with no arguments*, and *not* **static**. Thus, the faculty member or student can simply add member functions with these properties to the "problem set class" and buttons will appear to execute the functions when the program is run.

The "problem set application" that manages the "problem set class" also automatically provides a console window with robust error-checked I/O and a graphics output window for the display of simple graphics.

2. OVERVIEW OF THE PROBLEM SET FRAMEWORK

In this section of the paper, we will discuss the origins of the Problem Set Framework and its evolution. This will lead to a list of goals and specifications for the behavior of the framework. In later sections of the paper, we will explain how to use the framework and discuss some aspects of its implementation.

2.1 Classic Problem Sets

In the past, when we taught the introductory courses in Pascal and later in C++, we developed a large collection of simple exercises students could do to practice the rules of arithmetic, loops, decision statements, working with arrays, and building and using helper functions. Most of these exercises required fewer than 15 lines of code, needed some input from the user, and usually printed some results along the way. In some problems, there was simple graphics output. For example, we have a problem asking students to paint ten circles across the graphics window.

In this situation, it was relatively easy to add a new problem to a problem set. One would create a new function that performed the problem task and any associated helper functions. It was then necessary to call the function in the switch statement used to select the next task to perform. In addition, some prompting code would also have to be written. The user interface was entirely through the console although the program had access to a graphics window for the display of simple vector graphics. The student was constantly prompted for what to do next.

2.2 Problem Sets in Java?

Moving to Java has given us the opportunity to concentrate on objects from the beginning and to use full graphical user interfaces in the large laboratory projects. However, students, especially the weak ones, still need basic practice in writing the small five to fifteen line functions that perform simple tasks and illustrate the use of elementary programming patterns. Manually creating a graphical user interface for such small problem sets seems like overkill. At the same time, using a fragile console driven interface with no graphics support seems too restrictive. We want the best of both worlds: ease of construction of the problem set combined with robust IO and graphics output.

The JPT toolkit already provides robust support for safe and user friendly I/O through the **console** object constructed in the **ConsoleAware** interface. To access this **console** in any class, the class must simply declare that it implements **ConsoleAware**. There are no additional steps that need to be taken. The JPT toolkit also defines a **BufferedPanel** class that provides graphics pane objects that will automatically refresh should the panes be hidden and then become visible. Thus we have the ingredients to build a

problem set framework but there is still work to do to figure out how to make everything automatic.

We want the actual problem set part of the program to be as simple as possible so that an instructor can easily add a new problem on the fly, in the live classroom, during the lecture. Similarly, we want students to able to add their solutions to the problem set and then test immediately without adding so much as a prompt string. We also want students to feel so comfortable with the framework that they can use it as a testbed to clear up any conceptual issues they may have by writing simple experimental code.

2.3 Goals and Specifications for the Problem Set Framework

As we struggled with the design of the Problem Set Framework and built early versions of the framework, our goals became clear:

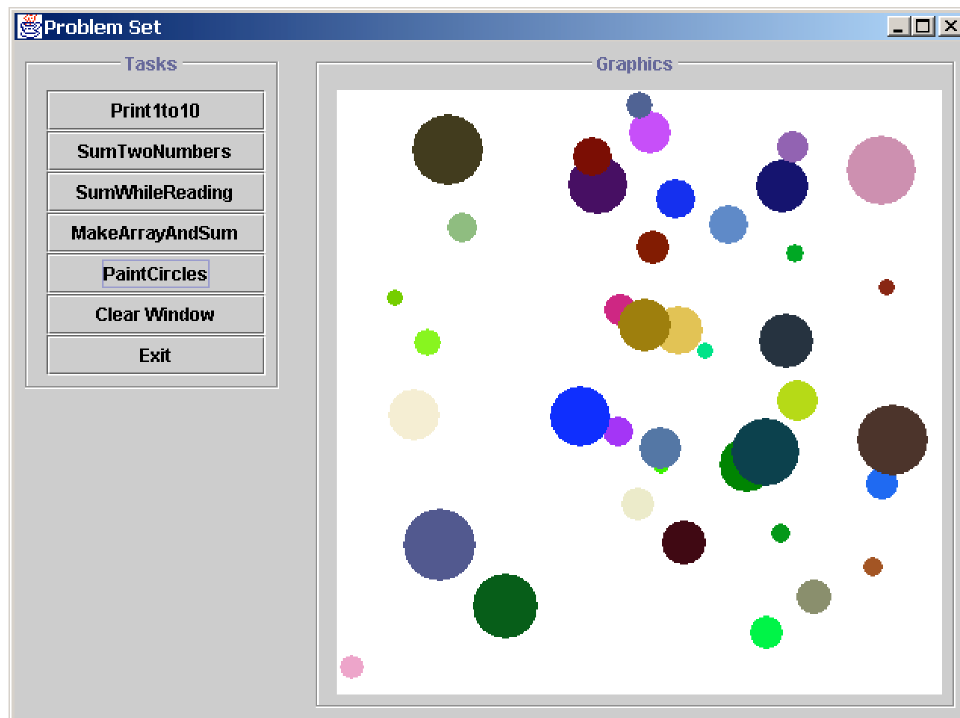
- each task in the problem set should be represented by one public function with no arguments
- any additional helper functions should be private or protected
- the application should automatically generate an action button for each public function in this set
- the label for the actions button should be related to the name of the task
- the actions should be synchronized, so that we cannot start a new task until the previously selected task has completed its execution
- the functions should have full access to the console without needing to add anything to the code of the class
- the functions should have full access to a graphics window again without needing to add anything to the code of the class
- there must be an action button that allows the user to clear the graphics window at any time
- there must be an action button that allows the user to terminate the program in case some task gets caught in an infinite loop
- there should be little or no additional code one needs to write to run the problem set
- it should be possible to give the problem window set an appropriate title

2.4 Initial Reaction to the Problem Set Framework

The framework we created was an instant success. Two other instructors teaching the introductory course started using it immediately in the classroom. The collection of C++ algorithmic exercises was adapted for use with Java within two weeks. The students were enthusiastic about having access to sample solutions on-line and being able to try a number of problems on their own.

3. USING THE PROBLEM SET FRAMEWORK

The *Problem Set Framework* comes with several sample methods that the student may examine to understand how to use the framework. Here is a screen snapshot of the GUI window after the method *PaintCircles* has been executed.



Before we examine how *PaintCircles* works, let us look at some of the simpler methods. To create the button *Print1To10* and its functionality, all that is required is to define the following method in the **ProblemSetClass**.

```
    public synchronized void Print1to10() {
        for (int i = 1; i <= 10; i++)
            console.out.print(i + " ");

        console.out.println("\n");
    }
```

The framework automatically encapsulates this method into an action that will be executed by a button named *Print1To10* that is placed into the button area of the GUI window. Similarly, to create the button *SumTwoNumbers* and its functionality, all that is required is to define the following method in the **ProblemSetClass**.

```
    public synchronized void SumTwoNumbers() {
        int x = console.in.demandInt("Enter x:");
        int y = console.in.demandInt("Enter y:");

        console.out.println("Sum: " + (x + y) + "\n");
    }
```

In both examples, the simple definition of a method that is **public**, **synchronized**, **void**, *with no arguments*, and *not* **static** is enough to cause the definition of a corresponding

button in the GUI window that will execute the method. These two examples use the JPT **console** for input-output. The **console** will be discussed in detail below.

The method **PaintCircles** is more elaborate but that is primarily due to the necessity to make some calls to Java2D graphics [3].

```
public synchronized void PaintCircles() {
    // do graphics setup
    window.clearPanel();
    window.repaint();

    Graphics2D G = window.getBufferGraphics();

    Ellipse2D.Double E = new Ellipse2D.Double();

    // request the number of circles from dialog
    int n = demandIntFromDialog
        ("How Many Circles?", "PaintCircles", "100");

    // draw filled circles
    for (int i = 0; i < n; i++) {
        // define diameter with 10 <= diameter <= 50
        int diameter = 10 + (int)(40 * Math.random());
        int maxX = window.getBufferWidth()  - diameter;
        int maxY = window.getBufferHeight() - diameter;

        // define top-left corner
        int x = (int)(maxX * Math.random());
        int y = (int)(maxY * Math.random());

        // define circle
        E.setFrame(x, y, diameter, diameter);

        // define r, g, b color
        int r = (int)(255 *  Math.random());
        int g = (int)(255 *  Math.random());
        int b = (int)(255 *  Math.random());

        // fill the random circle with random color
        G.setPaint(new Color(r, g, b));
        G.fill(E);
    }

    // refresh
    window.repaint();
}
```

The **window** object that corresponds to the graphics area in the screen snapshot is a Java Power Tools **BufferedPanel** object that maintains its data in an internal **BufferedImage** that is used to refresh the screen image when required. Since this **window** object delivers a standard **Graphics2D** object for painting, all of Java2D graphics is available. The only other tool in this example that is not pure Java is the method **demandIntFromDialog** that is packaged with the *Problem Set Framework*.

The purpose of discussing the three examples *Print1To10*, *SumTwoNumbers*, and *PaintCircles* in depth is to emphasize how easy it is to install new problems into the *Problem Set Framework*. In classrooms situations, teachers build such examples on the fly to provide simple demonstrations or to respond to student questions. Outside of class, students use the framework to answer their own questions or to test code in

their projects. In addition, several of our laboratories use the *Problem Set Framework* explicitly to test algorithms or class definitions.

3.1 Creation of a Problem Set

In general, we define an application class **ProblemSetApplication.java** that for a given problem set class **ProblemSetClass.java** automatically creates an entire graphical user interface with an action button for each “problem”. The action buttons are labeled with the names of the functions that implement the tasks. In addition, the programmer has access to fully error-checked I/O through the JPT **console** class and has access to creating graphical images in the attached buffered graphics panel **window**. The **ProblemSetClass** class also has a **title** member variable that may be used to set the main window title and an **application** member variable that may be used to access the main application object if needed.

To define a member function of the **ProblemSetClass** class that will automatically generate a button in the graphical user interface, the member function should be declared as **public**, **synchronized**, **void**, *with no arguments*, and *not* **static**. All other member functions in the **ProblemSetClass** class will be considered as hidden *helper functions*. Thus, for example, to ensure that a function is considered as a helper function it is enough to declare it as **protected**. It is also possible to define *helper data variables* in the **ProblemSetClass** class if needed.

Let us explain the reason for the conditions specified for those *member functions that will automatically generate a button*. If a simple button is clicked there is no obvious mechanism to supply arguments or use return values so we specify that the function must have zero arguments and be **void**. The function should be **public** to be visible in the application class. The function *should not be* **static** since in that case it cannot access the **window** variable. Finally, the function needs to be **synchronized** because otherwise if multiple buttons are clicked at once then the output from one button can become mixed into the output of another button in the console or graphics windows.

Thus, to summarize, to define a member function of the **ProblemSetClass** class that will automatically generate a button in the graphical user interface, it is enough to define the function as **public**, **synchronized**, **void**, *with no arguments*, and *not* **static**. Absolutely no additional effort is necessary.

3.2 Usage of the **console**

The **console** is an object in the JPT toolkit that controls three streams of data, the **in** stream, the **out** stream and the **err** stream, in a manner similar to the generic Java facilities provided by the **System** class. However, the **console** has several advantages over the system supplied console window. First, the program can elect to display the three streams in different colors. This permits students to easily distinguish what part of the display was generated by normal output (black), what part constitutes user input (blue), and what part was generated through the error handlers (red).

Some operating systems create a fixed size console for a Java program. Another advantage of our **console** window is that it scrolls automatically to allow the user to see the entire transcript of the program-user interaction. Moreover, when the **console** is closed, the user is offered the option of saving the entire transcript to a file, another feature that supports good pedagogy.

The best feature of the **console** is the way it handles user input. The JPT provides input functions that specify the type of input expected and subsequently perform

automatic error checking and error handling. These functions will deliver only valid data as a result of a request for input.

Input functions from the console use the prefix **console.in.** followed by a particular input function. Let us, for example, look at integer input. There are two forms:

```
int x;            // the variable that will get the value
int d = ...;      // some default value to be used in form 2

// form 1 with no default
 x = console.in.demandInt(prompt);

// form 2 with a default value d
 x = console.in.demandInt(prompt, d);
```

Here **prompt** represents a **String** argument that will be printed in the **console** to signal to the user that input is being requested.

In the *demand functions*, the user *must supply valid data*. If the data has an error, the demand functions will continue to prompt until the error is corrected. Thus, the program can be guaranteed that it will receive valid data to work with. In the case of form 2, if the user simply presses return, then the default value **d** will be returned. There are demand functions for each of the primitive types and for the **String** type.

In the Java Power Tools, there is the general notion of a **Stringable** type that means a type whose data state can be encapsulated into a **String** or conversely be set using a **String**. There are **Stringable** types **XByte**, **XShort**, **XInt**, **XLong**, **XFloat**, **XDouble**, **XChar**, **XBoolean**, and **XString** built into the JPT. For **Stringable** objects **X**, there are demand functions that have the following method signatures:

```
public void demand
    (String prompt, Stringable X);

public void demand
     (String prompt, String default_data, Stringable X);
```

Thus, the methods calls on **X** would look like:

```
// form 1 with no default
 console.in.demand(prompt, X);

// form 2 with default data
 console.in.demand(prompt, default_data, X)
```

In many instances, you would like *to prompt the user for data but if the user has no more data to supply then the program should simply move onward*. The way to accomplish this in the JPT is with the **reading** function that returns **true** if the user did supply data and **false** otherwise. The best way to understand this function is through an example.

```
XInt X = new XInt();

 while(console.in.reading(prompt, X)) {
    int x = X.getValue();
```

```
        // now do what you want with the int value x
    }
```

In this example, the loop continues precisely as long as the user supplies data for **X**. Inside the loop, the ordinary int **x** is extracted from the object **X**. The loop stops when the user simply presses return at the prompt.

The **reading** method is the basis for the *SumWhileReading* example in the framework.

```
public synchronized void SumWhileReading() {
   int sum = 0;
   XInt x = new XInt();

     while (console.in.reading("Enter value to sum:", x))
        sum += x.getValue();

     console.out.println("\nSum: " + sum + "\n");
}
```

Below is an extract of the **console** interaction when this method is used to sum four numbers. Notice that the *Problem Set Framework* will automatically print a message when the method begins and when it ends.

```
Run: SumWhileReading

Enter value to sum: 3
Enter value to sum: 1
Enter value to sum: 6
Enter value to sum: 5
Enter value to sum:

Sum: 15

End: SumWhileReading
```

The user terminates the input sequence by pressing return at the prompt. This is similar to the use of end-of-file to terminate input in traditional applications but is much more flexible since the computer-user dialog may actually continue later on.

To conclude this discussion of the **console**, there are three functions that control input-output flow rather than specifically supply data. These functions look like:

```
// ask a yes-no question and return true-false as a result
// use confirm in an if-statement or a while-statement

  if (console.confirm(question, default_response)) .... else ....
or
  while (console.confirm(question, default_response)) ....


// print standard prompt and wait until the user presses return
console.pressReturn();


// print supplied prompt and wait until the user presses return
console.pressReturn(prompt);
```

3.3 Usage of the graphics **window**

The graphics is displayed in a JPT **BufferedPanel** named **window**. For introductory exercises, we mostly care about the fact that any Java2D Graphics methods may be used to modify the screen image. There is a wealth of interesting exercises students can do with the use of graphics such as painting scalable drawings; painting a series of geometric objects in a given pattern; illustrating array data as bar charts; simulating Brownian motion; etc. The key point is that the **window** object can deliver its *graphics context* so that further graphics work may be done:

```
Graphics2D G = window.getBufferGraphics();
```

With this object **G**, all of the functionality of Java 2D Graphics is available.

4. DESIGN OF THE PROBLEM SET FRAMEWORK

We describe the evolution and the design of the Problem Set Framework. This will show how we arrived at the functionality by gradually addressing new concerns at each stage. This section of the paper may be used in an object-oriented design course as a simple case study that illustrates the uses of some important features of Java in a compelling yet ultimately simple example.

4.1 Persistence of the **console** and the **window**

The GUI panel with the buttons and graphics **window** is created in one Java frame and the **console** in another Java frame. It would be possible to create a new **console** for each exercise and close it on completion, but that seems like a very cumbersome way of doing things. If a student wants to run the same task several times in a row, this approach would erase the previous results and start with a clean slate for each repetition of the same task. We believe that it is important that the **console** remain active and visible while the application is running so that the student can look at the accumulated feedback. Therefore, the **console** is activated by the main application and all tasks share in its use.

4.2 Evolution of the GUI and the Problem Set Application

The basic goal is to create a new action button for each task in the Problem Set Class in such a way that when the action button is pressed the appropriate task function will be performed. In the initial version of the framework, the communication between the console and the GUI regarding the focus of control went haywire. The GUI froze while the actions were performed. To solve this problem, each action had to be given its own thread. This was accomplished using a general wrapper class **ThreadedAction** that causes any Java **Action** to be executed each time in a new separate **Thread**.

Now the situation was better. Each task executed as it should with the GUI remaining active while the task ran. However, we soon realized that while one task was running, another could commence and that they would compete for control of the **console**. This meant that output and interactions in the **console** would occur in some random order. We solved this problem by making each task into a **synchronized** method in the **ProblemSetClass**. This meant that even if several buttons were pressed in quick succession, only one task thread could execute at a time and all other task threads would be blocked while waiting for the executing task to complete.

At this point we had the structure of the interactions as we wanted it, however, all action buttons were labeled *task1*, *task2*, etc. and all function performing tasks had names **"task1"**, **"task2"**, etc. listed in a separate table of strings. We wanted to use meaningful names for the tasks and we wanted to eliminate the separate table of strings since we knew that requiring a student to both construct a method *and* put a string into a table was a bug waiting to happen.

4.3 Automating the Creation of Actions and Buttons: Java Reflection

The Java *reflection package* provides mechanisms for examining classes and objects in detail. In particular, one can extract the declared methods for an object, determine both the names and the properties of these methods, and execute these methods by indirect means. We decided to use Java reflection to automate the process of making actions and their buttons from the tasks in the **ProblemSetClass** object **problemSet**.

The first issue to be faced is how to determine if a **Method** object **method** extracted via Java reflection is in fact **public**, **synchronized**, **void**, *with no arguments*, and *not* **static**. This is solved by the following static test function **isProperMethod**.

```
protected static boolean isProperMethod(Method method) {
    if (method == null)
        return false;

    if (method.getReturnType() != void.class)
        return false;

    if (method.getParameterTypes().length > 0)
        return false;

    int modifiers = method.getModifiers();

    boolean OK =
        ((modifiers & Modifier.PUBLIC) != 0)
    &&
        ((modifiers & Modifier.SYNCHRONIZED) != 0)
    &&
        ((modifiers & Modifier.STATIC) == 0);

    return OK;
}
```

The next issue is how to turn a **Method** into an **Action** object that ultimately may be installed in the GUI. Here we need to use the Java reflection technique of indirect method invocation and we need to trap all possible exceptions. The code to accomplish the method-to-action task is:

```
protected Action makeActionFromMethod(final Method method) {
  if (method == null)
    return null;

  final String name = method.getName();

  return new ThreadedAction(
    new SimpleAction(name) {
      public void perform () {
        console.out.println("Run: " + name + "\n");

        try{
            method.invoke(problemSet, null);
        }
          catch (IllegalAccessException illAccEx){
          // should never happen
          // method is public
            console.err.print  ("IllegalAccessException: ");
```

```
            console.err.println("" + illAccEx);
            console.err.println("In: " + name + "\n");
        }
        catch (IllegalArgumentException illArgEx){
            // should never happen
            // method has zero arguments
            console.err.print  ("IllegalArgumentException: ");
            console.err.println("" + illArgEx);
            console.err.println("In: " + name + "\n");
        }
        catch (InvocationTargetException invTarEx){
            // may happen if exception thrown inside method
            console.err.print  ("InvocationTargetException: ");
            console.err.println("" + invTarEx);
            console.err.println("In: " + name + "\n");

            Throwable target = invTarEx.getTargetException();

            console.err.println("Target Stack Trace:");
            target.printStackTrace(console.err);
            console.err.println();
        }

        console.out.println("End: " + name + "\n\n");
      }
    });
}
```

The biggest complication in this call is that the indirect method invocation call **method.invoke(ps)** can in principle throw three kinds of exceptions and these must be caught here and dealt with. In our situation, the first two exceptions cannot happen since we know that the method passed is **public** *with zero arguments*. The final exception will be thrown if the method itself throws an exception. In this case, we print the stack trace for the exception as supplied by standard Java calls. Because we trap all exceptions that arise from method calls initiated in the **ProblemSetClass**, the program continues to run and the student may examine the error messages and then continue to test before exiting to make corrections.

The method call **method.invoke(ps)** is the heart of the **perform()** method of the object of the class **SimpleAction** that is created on the fly to encapsulate the method behavior as an action. This **SimpleAction** object is immediately passed to the **ThreadedAction** constructor to create a wrapper action that will always execute in a new separate **Thread**. It is this **ThreadAction** that is installed as a button in the GUI.

With the foundation of **isProperMethod** and **makeActionFromMethod**, it is now possible to use Java reflection to extract all methods in the **problemSet** object and create a list of **Action**'s that can then be used to create the buttons in the GUI. This work is accomplished by the method **getActionList**. The crucial Java reflection code in this method is:

```
Method[] methodList
    = problemSet.getClass().getDeclaredMethods();
```

The rest of **getActionList** is straightforward. We loop over the methods and create an action for each method that is proper. We then return the actions in an array that will be used to construct a JPT **ActionsPanel** that automatically creates the desired buttons.

4.4 Collaboration of **ProblemSetClass** and **ProblemSetApplication**

The **ProblemSetClass** and **ProblemSetApplication** are mutually referential. The constructor for **ProblemSetClass** reads:

```
public ProblemSetClass(ProblemSetApplication application) {
    this.application = application;

    if (application != null)
        window = application.getGraphicsWindow();
}
```

Notice that the **ProblemSetClass** object records the application object and extracts from the application object a reference to the graphics **window** in the GUI. The constructor for the **ProblemSetApplication** object begins by constructing an object **problemSet** of class **ProblemSetClass**:

```
public ProblemSetApplication() {

    // define the problem set object
    problemSet = new ProblemSetClass(this);

    // the rest of the constructor builds the GUI ...
```

Thus the **application** and the **problemSet** are able to mutually communicate. In particular, this is how the **application** object can extract the proper methods from the **problemSet** object to make the actions and then the buttons.

5. CONCLUSION

5.1 Our Experiences

We created this framework out of frustration with other techniques for creating simple problem sets. As soon as it became available it was enthusiastically adopted by faculty, students, and undergraduate tutors working with students. All felt that this framework provided a very simple yet robust environment for trying out the behavior of Java functions, control structures, arithmetic, and the class reference model. No Java features are hidden.

While we are firm believers in the power of graphical user interfaces for more complex programs in which the user needs to see the entire state of several objects while the actions are being performed, it is a great benefit to have a tool for examining the functionality of one simple action, one change in the object state, or one control structure (loop, decision statement) at a time. The framework can be used in any Java setting, even if the adopter is not interested in using the JPT for other projects. All one needs is to do is to compile together the **jpt.jar** and the **ProblemSetApplication** with the specific version of the **ProblemSetClass**.

5.2 Additional Pedagogical Applications: Testing and Demo Programs

Because the full power of Java is available in the *Problem Set Framework*, the proper methods in the **ProblemSetClass** class can in fact do anything including instantiating arbitrary objects and calling arbitrary methods on these objects. Thus, the Problem Set Framework is also *a tool for the creation of test suites for arbitrary classes*. One can instantiate a test object and then run any desired set of methods that needs to be tested and create any desired feedback either as text in the **console** or as graphics in the **window**.

In addition, all stand-alone Java applications have at least one class with a **main** method. Although this **main** method takes an array of **String** arguments, in most cases the array that is actually passed is **null**. Thus, it is possible to launch an arbitrary application **Foo** in the Problem Set Framework using code of the form:

```
 public synchronized void RunFoo() {
     Foo.main(null);
}
```

Thus, the *Problem Set Framework* can easily be used to create launch pad for a suite of demo programs. It has not escaped our attention that it is possible to use Java reflection to scan a family of classes looking for a **main** method and then to create for all such classes the equivalent of the **RunFoo()** method. However, we choose not to pursue this further here.

On a different note, much of the power of the *BlueJ* environment [4] comes from the fact that *BlueJ* can instantiate an object of any class and then make methods calls on that object. Those familiar with *BlueJ* will realize that a significant subset of what is done in *BlueJ* can now also be done with a small effort in the Problem Set Framework *in whatever Java development environment you choose to work in*. We have always felt that the Java Power Tools and *BlueJ* are synergistic and this is one more instance of the potential relationships.

5.3 Acknowledgements

Partial support for this work has been provided by the NSF grant DUE-9950829.

5.4 Dissemination

The Java Power Tools, *Problem Set Framework*, and related files may be found at:

http://www.ccs.neu.edu/teaching/EdGroup/JPT/

6. REFERENCES

[1] Bruce, K. B., Danyluk, A., and Murtagh, T. P., *A Library to Support a Graphics-Based Object-First Approach to CS1*, SIGCSE Bulletin, 33(1), 2001, 6-10.

[2] Cornell, G., and Horstman, C., *Core Java 1.2 Volume 1*, SunSoft Pres, Mountain View, CA, 1999.

[3] Knudsen, J., *Java 2D Graphics*, O'Reilly, Sebastopol, CA, 1999.

[4] Koelling, M., and Rosenberg, J., *Object First with Java and BlueJ*, SIGCSE Bulletin, 32(1), 2001, 429.

[5] Koffman, E., and Wolz, U., *A Simple Java Package for GUI-like Interactivity*, SIGCSE Bulletin, 33(1), 2001, 11 - 15.

[6] Lambert, K. A., and Osborne, M., *JAVA Complete Course in Programming & Problem Solving*, South-Western Educational Publishing, Cincinnati, OH, 2000.

[7] Raab, J., Rasala, R., and Proulx, V. K., *Pedagogical Power Tools for Teaching Java*, SIGCSE Bulletin, 32(3), 2000, 156-159.

[8] Rasala, R., Raab, J., and Proulx, V. K., *Java Power Tools: Model Software for Teaching Object-Oriented Design*, SIGCSE Bulletin, 33(1), 2001, 297-301.

[9] Roberts, E., *An Overview of MiniJava*, SIGCSE Bulletin, 33(1), 2001, 1-5.