Java Power Tools and Java Power Framework

Richard Rasala, Viera K. Proulx, Jeff Raab
College of Computer Science
Northeastern University
Boston MA 02115

{rasala,vkp,jmr}@ccs.neu.edu

http://www.ccs.neu.edu/jpt/

Introduction

One of the promises of object-oriented programming is to simplify program design by encapsulation of common properties and smooth handling of variations. In the domain of building graphical user interfaces in Java, however, this promise is not entirely fulfilled. Although the Java Swing libraries provide a plethora of tools and widgets, the programmer must still build a graphical user interface step-by-step with lots of low level repetitive code. The goal of the *Java Power Tools* library is to maximize the use of encapsulation, recursion, and reflection so that graphical user interfaces may be constructed extremely rapidly. These tools may be used by faculty to create user interfaces for freshman level exercises and by students themselves in upper level project courses. In addition, using the *Java Power Framework*, certain graphical user interfaces may be created entirely automatically with no explicit GUI programming whatsoever.

From the beginning, we intended the *Java Power Tools* to be not only a toolkit for building graphical user interfaces but also a design model for object-oriented design[1]. At present, the tools consist of 142 classes that illustrate a layered approach to design. The tools make significant use of the following patterns identified in the well-known book on *Design Patterns*[2]: Creational (Factory Method, Singleton), Structural (Composite, Adapter, Decorator, Facade), and Behavioral (Command, Memento, Strategy, Template, Observer, State).

In this paper, we do not plan to summarize *JPT* since that is done in [1] and other places. In line with the title of this workshop, "Killer Examples", we will focus on examples beginning with the *Java Power Framework* (*JPF*) that enhances our earlier *Problem Set Framework* [3] and is designed to enable "instant Java".

Java Power Framework

The *Java Power Framework* class `JPF` provides the foundation for creating an automatic Java GUI application with access to the *JPT* console I/O and a simple graphics window. To start, we must define a class that extends `JPF`. For convenience, we will call this class `Methods` although its name may be anything we wish.

```
import edu.neu.ccs.jpf.*;

public class Methods extends JPF { ... }
```

If this class will be used as the Java `main` program, then we should add the following definition of `main`:

```
public static void main(String[] args) { new Methods(); }
```

If some other class will act as the `main` program then that class simply needs to call `new Methods()`. Note that there is no need to define a constructor in `Methods`. Java will supply a default constructor and that constructor will in turn invoke `new JPF()` which will do all of the magic.

In this vanilla state, the invocation of `new Methods()` will bring up the *JPT* console window and will also bring up the automatic GUI shown in Figure 1.

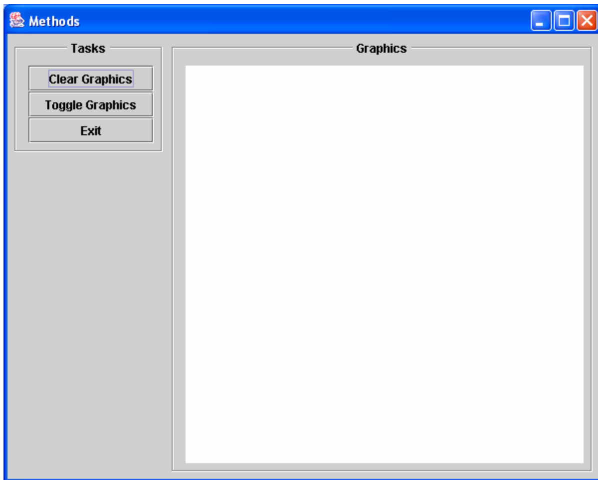

Figure 1

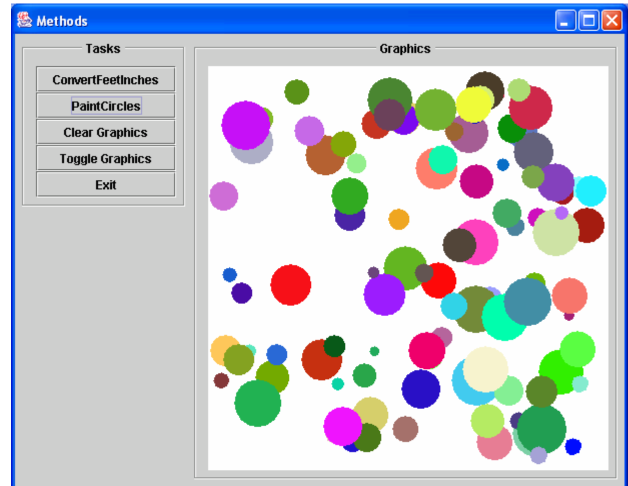

Figure 2

The three buttons in Figure 1 represent buttons that are always present in the automatic GUI. The first button will clear the graphics window, the second will hide or show the graphics window, and the third will call `System.exit(0)` to halt all processes. Suppose we now add two methods to the `Methods` class as follows:

```
public int ConvertFeetInches(int feet, int inches) { return 12 * feet + inches; }
public void PaintCircles(int circleCount) { /* graphics details omitted */ }
```

When we compile and execute, two new buttons appear in the automatic GUI (Figure 2) that are labeled by the names of the two methods, namely, `ConvertFeetInches` and `PaintCircles`. If the `ConvertFeetInches` button is clicked, then the following "method GUI" is displayed (Figure 3):

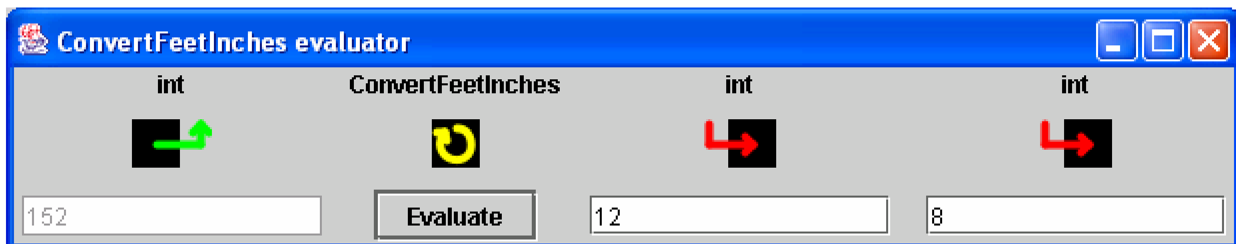

Figure 3

This "method GUI" allows the user to experiment with the method. Along the top row of this GUI are labels that remind the user of the method signature: `int ConvertFeetInches(int, int)`. The two text fields on the right side are for user input and this is signaled by the red arrow icons that enter the "black box" of the function. The "Evaluate" button under the function name is used to evaluate the function and this is signaled by a yellow arrow icon that circles within the "black box" to signify internal processing. Finally, the return value of the function is placed in the text field on the left side and this is signaled by the green arrow icon that exits the "black box". The return value field is disabled so that the user may not directly enter data into this field. If the user makes any typographical errors in entering the function parameters, this problem will be handled automatically by error dialogs generated by the *JPT*.

If the `PaintCircles` button is clicked, a similar "method GUI" is shown that allows the user to enter the desired circle count. When the "Evaluate" button is clicked, the method paints that many random circles in the graphics window. A typical result is shown in the graphics window of Figure 2.

In general, the action buttons of a *Java Power Framework* application are created automatically from `public` methods whose arguments and return type are one of the following types: `void`, `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`, `String`, `Color`, `BigInteger`, `BigDecimal`, or any type that implements the *JPT* `Stringable` interface. The creation of the automatic *JPF* application is accomplished by a combination of Java reflection and the use of the rest of the *Java Power Tools*. Since these techniques are encapsulated, the user of the *Java Power Framework* need not have any knowledge of the *JPT*. Thus this tool may be on the very first day of a course using Java.

The *Java Power Framework* is not restricted to handling just simple numerical or graphical examples. *The body of a method in the* `Methods` *class can do anything that is possible in Java itself.* Specifically, these methods may instantiate objects that are either used for the duration of the method call or saved for further use as member data in the `Methods` class. Once such objects have been defined, any of their methods may be called with any of the resulting return values becoming available for immediate use or for storage. Pedagogically, this capability may be used both by faculty for on-the-fly examples in lecture and by students for personal experiments outside of class.

After learning to use *Java Power Framework* for simple experiments with classes, students can then learn to use it for more systematic testing of classes. Using one or more methods in a typical `Methods` class, *it is possible to build an entire test suite for a single class or for a family of related classes.* A good pedagogical plan is for faculty to supply some tests and for students to supply additional tests. To support faculty tests, the `JPFBase` class provides helper methods that will test if another class declares or provides via inheritance a set of constructors and/or methods. If any such functions are missing, a dialog of the following form will be presented to the student.

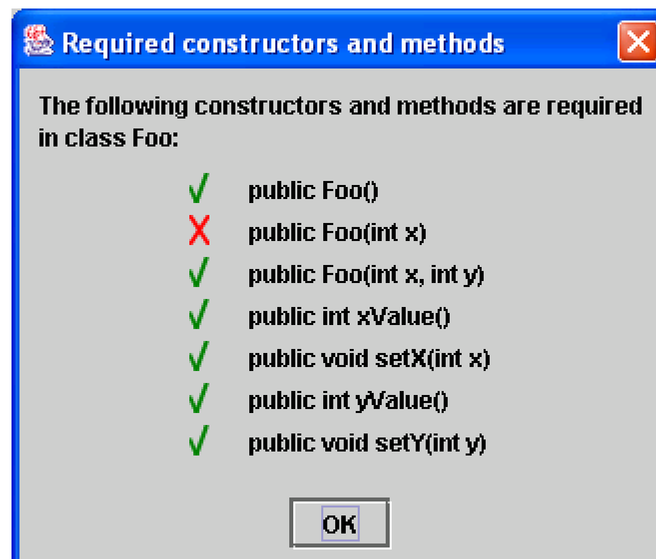

Figure 4

The student is therefore given immediate feedback about how much has been done and what is left to do.

In the *Java Power Framework*, a method may also launch other GUIs. These GUIs may be built directly using pure Java (AWT or Swing) or by using the rapid development tools in the rest of *Java Power Tools*. In particular, if we have a Java class `App` that launches its own GUI application, then we can create a *JPF* button to launch `App` as follows:

```
public void LaunchApp() { App.main(null); }
```

Many of our student labs build custom GUIs. As we update these labs this year, we plan to launch their GUIs using a "launch method" in a suitable *JPF* `Methods` class. This will permit us to set up appropriate unit tests for

the lab in the **Methods** class and will allow students to add additional tests for debugging purposes. This flexibility will provide students with an environment for seamless integration of development and testing.

The ability of a *JPF* **Methods** class to launch applications may also be used to build a *master application* that can launch any of the demonstration programs or lab exercises for an entire course. Then, in lecture, when students ask questions, the teacher has instant access to all of the course materials so that responses can be rapid and to the point.

Laboratories Built Using Java Power Tools

In this brief summary, we cannot describe the many labs available on the JPT web site but we will briefly describe one that is particularly special, namely, the *Automatic Array Algorithm Animation* lab (Figure 5).

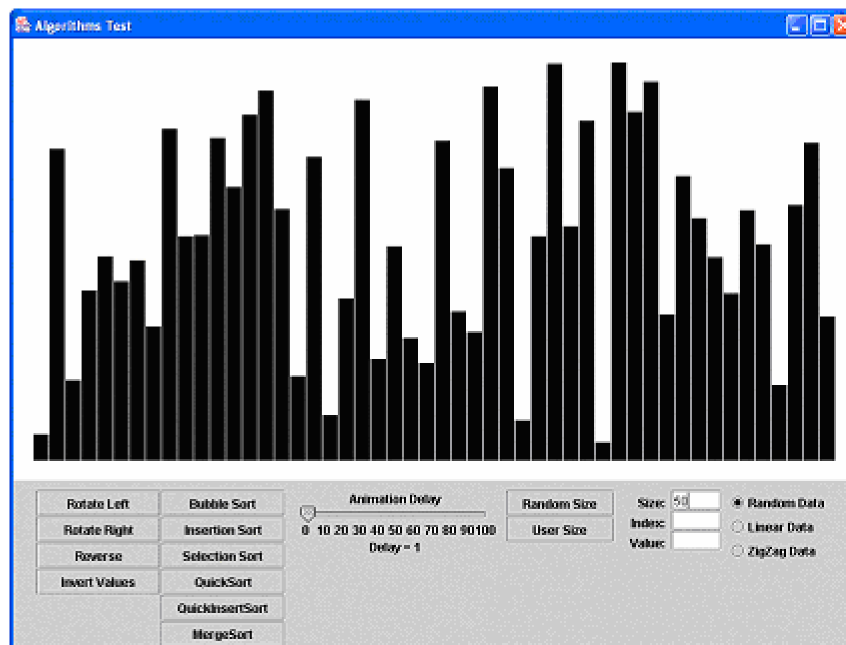

Figure 5

In this lab, the students encapsulate their array algorithms as action objects using a Template pattern that acts on a simple array class **IntArray**. To provide automatic animation, the "Algorithms Test" application executes the algorithms on an object of the **BarChart** class that is a derived class of **IntArray**. Each time the **setValue** method is called by an algorithm, the corresponding bar in the bar chart is updated. In addition, because the algorithms are encapsulated as objects, it is easy to apply suitable wrappers so that the algorithms run as separate threads while acting in a synchronized fashion on the bar chart to prevent mutual interference. This design utilizes many of the tools in the *JPT* to facilitate both the algorithmics and the creation of the graphical user interface of the "Algorithms Test" program.

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[2] Richard Rasala, Jeff Raab, and Viera K. Proulx, *Java Power Tools: Model Software for Teaching Object-Oriented Design*, SIGCSE Bulletin, 33(1), 2001, 297-301.

[3] Viera K. Proulx, Richard Rasala, and Jason Jay Rodrigues, *Simple Problem Solving in Java: A Problem Set Framework*, J. Computing in Small Colleges, 17(6), 56-70.