Submission Title
Java Power Framework

Contact Information
Richard Rasala
rasala@ccs.neu.edu
Viera K. Proulx
vkp@ccs.neu.edu
College of Computer & Information Science
Northeastern University
Boston MA 02115

Problem Statement
Many educators have searched for a simple environment in which to teach Java. In our view, such an environment should have the following characteristics.

- The student should have a direct means to build objects, execute methods, and perform *instant exploration*.
- The environment should support both console IO and graphical interaction.
- The design and creation of classes and methods should be integrated with their testing.
- The use of the environment should scale from as little as the testing of one method to as much as the integration testing of an entire application.

The *Java Power Framework* which is available as part of the Java Power Tools provides such a pedagogical environment.

Solution Overview
The *Java Power Framework* is a framework designed to promote *rapid experimentation* as well as *systematic testing of classes and applications*. Let us explain how the *JPF* works.

The *Java Power Framework* is designed to *automatically turn methods into buttons*. A user can define as little as one method and then automatically get a button to execute that method. A user can also define dozens of methods and have buttons created for each that are placed in a scrolling panel. What is done in each method is up to the user but the possible uses include instant experimentation, on-the-fly creation of simple GUI frames, systematic testing of a set of classes and objects, and launching entire applications in order to perform integration testing.

The setup to use *Java Power Framework* is very easy. The user simply defines a class that extends the base class `JPF`. This extended class can have any name but we will call it `Methods` for this description. The `main` method then contains one line:

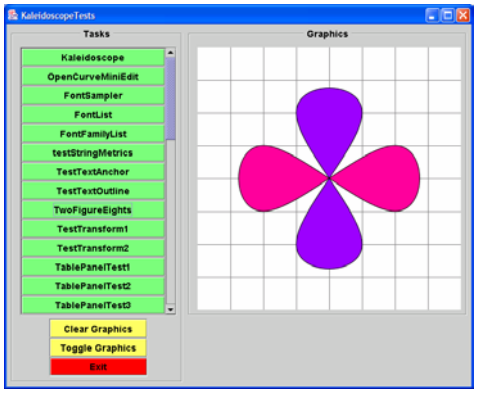
```
new Methods();
```

Normally, the `Methods` class does *not* define a constructor. The purpose of the call, `new Methods()`, is to invoke the base class constructor, `new JPF()`, and to inform `JPF` of the specific derived class `Methods` that has performed this invocation. The `JPF` constructor then scans the `Methods` class for *proper* methods that may be automatically instantiated as buttons in the *JPF GUI*. `JPF` also creates a panel for graphics experiments and opens a separate *console frame* for simple user interaction.

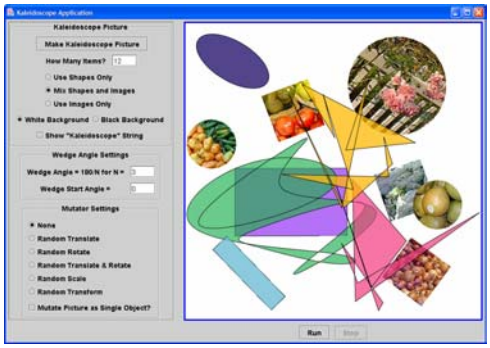
What is a *proper* method, that is, a method that may be attached to a button automatically? The simplest example is a method that is *public, void, with no arguments*. This covers a lot of ground since such a method can orchestrate a console dialog; can create objects and execute test methods on those objects; can define frames and hand off execution to those frames, or can even launch entire applications. Thus, the facility to turn methods into buttons is extremely powerful.

More generally, in *JPF*, a *proper* method is permitted to have arguments and/or a return value provided that the types involved are *simple*. By a *simple* type, we mean a type whose data values may be entered by the user using a single line of text input. The simple types include all of the primitive types, String, BigInteger, BigDouble, and Color. It is possible to define additional simple types using the JPT `Stringable` interface.

On the JPT 2.3 web site, there is a case study called Kaleidoscope. This case study was developed within the *JPF* so it is a good illustration of the power of the framework. Below is a screen snapshot of the automatic GUI created for this case study.



Notice the topmost button in the scrolling button panel. This button, named *Kaleidoscope*, launches the full Kaleidoscope application in its own frame.



How is the *Kaleidoscope* button created in the *JPF GUI*? Simply by adding the following method to the test class.

```
public void Kaleidoscope() {
  KaleidoscopeApplication.main(null);
}
```

There is absolutely no explicit GUI code in the framework to go from defining such a method to creating its associated button.

The *Kaleidoscope* button lets us perform integration testing of the nine classes in the Kaleidoscope project and the many base classes in the JPT that were added to support such programs. The remaining buttons in the *JPF GUI* do all sorts of tests from very simple unit tests to miniature applications that test modest pieces of the functionality used in the Kaleidoscope. For example, the button *TwoFigureEights* (whose output is shown in the first screen snapshot) executes a unit test to verify the rendering of the smooth cubic shapes being created by the JPT shape tools. This was one of the earliest tests in the Kaleidoscope project.

In *JPF* one can try simple experiments with either text or graphics output, do unit tests of a single class, test the interaction of several classes, run miniature applications, and run complete applications. This makes the *JPF* an extraordinary pedagogical tool.

Let us give some typical scenarios that illustrate how the *JPF* may be used in teaching. In the first scenario, imagine that a student is learning how to paint a circle. The student might define a simple test method:

```
public void PaintCircle
(double x, double y, double r, Color c) { ... }
```

In this method, x, y represents the center of the circle, r its radius, and c the color to paint. When the *JPF* button *PaintCircle* is clicked the following frame will be opened automatically:

The student may then experiment with various values of x, y, r, and the color c. The color may be input by clicking the color swatch which will open a JColorChooser dialog; by using r, g, b, alpha values in decimal or hexadecimal; or by using a dropdown list of named colors.

This scenario may be extended if the student is asked to create a **`Balloon`** class in which the rendering code is based on what was learned in the **`PaintCircle`** experiments. The student can then be asked to explore defining a *collection* of **`Balloon`** objects and animating these objects if one or more balloons are in motion.

More generally, the *JPF* supports the development of interacting classes. In a simple scenario, the student may be asked to define a **`Person`** class and an **`Address`** class. The exercise may be to permit the **`Person`** object to have zero or more **`Address`** objects in some internal structure. Here *zero* corresponds to a person whose address is unknown and *more than one* corresponds to a person who may have a home address, campus address, etc. In testing these classes, the student has the option to build all needed objects within each particular test method or to use some sample data that is set up as member data in the **`Methods`** class and is used by many test methods.

To enable testing with large data sets, it is possible to add a protected method **`loadData`** to the **`Methods`** class that arranges to load data from a local file, a web site, or a database. If desired, such a method can launch a dialog box that gives the student the choice of the current data set. In any case, to ensure that the data is available for all test methods, the student must simply modify the one line in the **`main`** method to read:

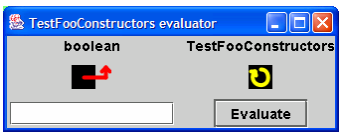
```
new Methods().loadData();
```

The design of the *JPF* permits *faculty defined tests to be included with tests created by students*. To accomplish this, the instructor provides a class **`Tests`** that extends **`JPF`** and then requires that the student class **`Methods`** extends **`Tests`**. Once this is done, the proper methods in both **`Tests`** and **`Methods`** lead to buttons in the *JPF GUI*.

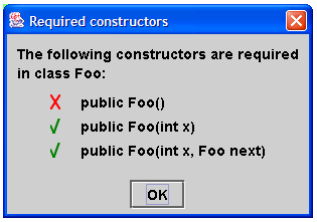
Among the tests that an instructor may supply are tests that check if a given student class **`Foo`** has required constructors or methods. For example, to test if the class **`Foo`** has certain constructors, the instructor might use a method of the form:

```
public boolean TestFooConstructors() {
  return declaresConstructors(Foo.class,
    new String[] {
      "public Foo()",
      "public Foo(int x)",
      "public Foo(int x, Foo next)" } );
}
```

When the *TestFooConstructors* button is clicked in the *JPF GUI*, it will open an evaluation frame.



Then, if the *Evaluate* button is clicked, it will show *true* if all of the required constructors are present and will open an auxiliary dialog if some constructors are missing:



In this example, the dialog signals that the default constructor for **`Foo`** is missing. When the user clicks *OK* then the original evaluation frame shows *false*.

We conclude this description of *JPF* by comparing the *JPF* testing framework to the well known JUnit tools. We believe that *JPF* is both simpler to use than JUnit and also more powerful. In *JPF*, all proper methods turn into buttons that may be clicked to execute their code. There is no need to name methods **`test...`** or to think of a **`TestCase`** class with **`run`**, **`setUp`**, and **`tearDown`** methods. Each proper method in *JPF* does exactly what it wants to do and calls what it wants as helper methods.

Fortunately, there is no need to choose between *JPF* and JUnit. If desired, JUnit tests may be set up and run alongside tests in *JPF*. We see JUnit as best suited for packaged tests that are run in a hands-off fashion. We see *JPF* as best suited for interactive tests that require the hands, eyes, and minds of the student to be fully engaged.

Experience with the Solution

The *Java Power Framework* is so simple that it may be used by students in the first week of class. To make things even easier, we provide a vanilla class that contains the shell of a **`Methods`** class, a **`main`** method, and a list of common Java imports. This shell enables students to get right to work on doing Java.

In a classroom setting, we typically keep a *JPF* project open and ready for use. At the beginning of the course, we may build both the sample classes and their test code in *JPF* on the fly. If the students make suggestions, these ideas can be incorporated and tested immediately and everyone can see what works and what doesn't. If someone asks a question, a side method can be built that performs an experiment that will answer the question. In this way, we model for students a programming style that encourges both experimentation and test-driven development.

For student laboratories and homework assignments, we require the use of the *JPF* so that simple tests, systematic unit tests, and full integration tests may be combined in a single framework.

API Documentation & Related Materials

The main JPT site to access documentation, code, and the jpt.jar:

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