Java Power Tools: Principles, Structure, Highlights

Richard Rasala, Viera K. Proulx
College of Computer and Information Science
Northeastern University
Boston, MA 02115
{rasala,vkp}@ccs.neu.edu
http://www.ccs.neu.edu/jpt/

ABSTRACT

The *Java Power Tools* is an extensive pedagogical toolkit that is designed for rapid development of graphical user interfaces, for robust IO with automatic error checking, for graphics display of mutatable shapes, images, and text, and for quick construction of all associated data structures. The Java Power Tools includes a framework, *Java Power Framework*, that is designed for efficient experimentation and testing. User methods defined in a derived class of the JPF class will automatically give rise to buttons in the framework that will execute the methods.

This article will summarize the pedagogical and design *principles* at the heart of the Java Power Tools, will discuss the *structure* of the JPT that permits these principles to echo throughout the various classes in JPT, and will *highlight* the most important JPT features of interest to faculty and students.

The Java Power Tools are built directly on pure Java and so may be used as a library within any Java development environment. In addition, JPT is 100% open source and so may also be used as a design model in upper-level courses on object-oriented design and software development. With the recent creation of an ACM Task Force to consider a standard *for pedagogically oriented APIs*[51, 52], it is an especially good moment to review what JPT has attempted and accomplished.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented programming.

Keywords

Java, frameworks, graphical user interfaces, IO, testing, eXtreme encapsulation, CS1, CS2, object-oriented design.

1. Introduction

Before beginning the design of the Java Power Tools, we had extensive experience with creating tools for IO and graphics in Pascal and C++. When our college decided to use Java in the first year courses, we in turn decided to create a first class set of tools for building graphical user interfaces in Java. Our goals were to build tools that combined power with simplicity so that it would be possible to build GUIs quickly and with no annoyances.

This work was partially supported by NSF grant DUE-9950829.

There is often a tension between simplicity and power. If a set of tools is poorly designed, then it may achieve power at the expense of excess complication. Indeed, many vendor tools such as those supplied with Java have just this property. It is possible with these tools to build graphical user interfaces, for example, but the effort is hard and the number of ideas and details that must be mastered is great.

It turns out to be much, much easier to build GUI's using Java Power Tools. This shows that the problem of building GUI's is not in itself complex. Let us describe the design principles that have led to the combination of simplicity and power in the Java Power Tools.

The first design principle is to use *eXtreme encapsulation*[42] to hide all technical details that may be deduced from the high level choices being made by the user of a class. JPT provides a variety of constructors to build GUI objects in a mature state, that is, ready to use by the program without further tweaking. Within these constructors and their support methods, the repetitious code typical of standard Java GUI programming is encapsulated.

The second design principle is to enable the *recursive composition* of simpler objects into more complex objects that in turn may be combined into even more complex objects. In particular, we use a functional/declarative programming style whenever possible.

The third design principle stems from remarks of Alan Kay[22]. He has described objects as *sites of higher order behaviors*. This suggests that we should think of objects primarily in terms of their behaviors rather than in terms of their data. This viewpoint led us to our third design principle: *encapsulate actions as objects*. We attach behavior to components by attaching appropriate action objects. This approach implies that the traditional model-view-controller model for GUI's is replaced by model-view-action. There is no need to have a controller object if the appropriate action objects are properly attached to the visible elements in the GUI view. In terms of design patterns, the principle *encapsulate actions as objects* amounts to the systematic use of the *strategy pattern* throughout the JPT library.

Having briefly described the fundamental design principles of the Java Power Tools, we will now discuss the highlights of JPT and along the way elucidate its structure.

2. Java Power Framework

The Java Power Framework is a framework designed to promote *rapid experimentation* as well as *systematic testing of classes and applications*. The JPF is so easy to use and so flexible that it is fair to say that if a faculty member were to choose to use only JPF and no other part of JPT then this would still be of great benefit to students. Let us explain why JPF is so powerful.

The Java Power Framework is designed to *automatically turn methods into buttons*. A user can define as little as one method and then 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` program 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 instantiated as buttons in the JPF GUI. In this automatically constructed GUI, JPF also creates a panel for graphics experiments. Finally, JPF separately opens a 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, or can create objects and execute test methods on those objects, or 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, 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 a 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.

If a proper method has arguments and/or a return value, then it is not executed directly when its button is clicked. Instead a new frame is opened in which the user can enter input values for the arguments, then click a button to execute the method, and finally view the return value. This auxiliary frame will stay open for performing multiple tests until it is closed explicitly by the user.

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` program, 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[3]. We want students to see that if they are puzzled then they can answer some of their own questions immediately and not simply have to wait to ask a teacher or another student.

We have mentioned that JPF provides a panel for graphics experiments and access to console IO. We will describe these features later since they are generally available in JPT.

We conclude this description of JPF by comparing the JPF testing framework to the well known JUnit[21] tools. We will argue that JPF is both simpler to use than JUnit and also more powerful. In JPF, all proper methods will turn into buttons that may be clicked to execute their code. There is no need to name test methods `test...` or to think of a `TestCase` class with `run`, `setUp`, and `tearDown` methods. Each proper method in JPF does what it wants and calls what it wants as helper methods.

JPF also encourages a broader range of testing than JUnit. JUnit tests focus on testing that methods return the correct values when given arguments that should produce known results. This is a very important aspect of testing but it does not test issues such as: does a graphics method produce the correct graphics output? does a class create a GUI frame that has the desired layout? does an entire application integrate its components and algorithms so that it behaves as expected interactively? These test issues are beyond JUnit but are handled easily in JPF.

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 user to be fully engaged.

3. Input-Output

A fundamental design goal of the Java Power Tools is to provide input-output facilities that:

- Work, as far as possible, the same way in the console or in a GUI.
- Support the most common data types directly.
- May be extended to user-defined data types.
- Contain robust error detection and correction.
- Permit the use of input filters to apply constraints.
- Support the automatic evaluation of mathematical expressions in user input data.
- Enable two input styles: a *demand* mode in which the user must supply input and a *request* mode that allows the user to cancel a pending input operation.
- Are really easy to use.

These goals have been achieved and let us describe how this was accomplished.

In order to have a general, extensible mechanism, we needed to deal with input of objects and then derive input for the primitive types using adapter methods. The input of objects is based on the JPT `Stringable` interface which requires that the object be able to set its state from a suitable text string; that it be able to produce a text string from which its state may be reset; and that it have a default constructor. The decision to base IO on text even in a GUI setting was inspired by a comment of Berners-Lee[6] that only text is a *basic lingua franca*. We briefly considered using XML as part of our format but decided that XML would not work with direct user input and therefore would be more of a barrier in our code than a help.

Having defined the **Stringable** inteface, we then built a set of base **Stringable** classes for the 8 primitive types and for String, BigInteger, BigDecimal, Color, and Point2D. We also built mechanisms to enable user defined **Stringable** types.

Modifying the state of a **Stringable** object depends on passing a string to its **fromStringData** method. Of course, if this string is invalid, the method must do something so it throws a parse exception that can be caught and handled in the IO routines.

The **fromStringData** method provides a point of leverage to add bells and whistles to the parsing. This is how we are able to support automatic evaluation of mathematical expressions for numeric types and a number of special formats for Color input. It is amazing how much easier it is to do input with the ability to write input expressions such as *exp(2)* or *(1 + sqrt(5))/2*.

We reach the remaining goals for our input-output facilities by the design and structure of the IO routines. At the top, we support both **demandObject** and **requestObject** to mirror the two styles of input. Since the user may cancel a *request*, such a method will throw a cancelled exception that may be caught by the application to divert further processing. It is important that error detection and recovery are handled automatically within the *demand* and *request* methods so that the caller is assured of valid data without the need for ad hoc error processing.

Both *demand* and *request* methods come in two forms, one that is plain vanilla and one that permits the caller to pass a filter. Thus, filtering is available but is not forced on the caller. There are a number of classes to build standard filters and to combine filters. It is also possible to easily create a filter on the fly.

Once the fundamental IO structure was in place for objects, we patiently supplied convenience methods of the same kind for the 8 primitive types and String, BigInteger, BigDecimal, and Color. Our motto is: *Support the general case and then support the special case too*.

We achieve compatibility between console IO and GUI IO by calling most of the same underlying routines. There are of course a few differences arising from the fact that a console is only a stream of text while a GUI has auxiliary IO mechanisms such as buttons. Thus, a *request* method is implemented in the console by interpreting an empty input string as a cancel operation. In a GUI, we can more directly provide a cancel button. When an error occurs in console input, we print the error message and then ask for the input again. When an error occurs in a GUI, we highlight the field in which the error occurs and we bring up a dialog box in which the user can correct the error or cancel.

In our view, the JPT input facilities address all of the concerns that have been mentioned by academics about Java IO and then add features and flexibility that many have not even dreamed of. To emphasize this, let us do two simple examples.

Imagine a **Person** class with the following constructor:

```
public Person(String name, int age)
```

Then here is how we can define a **requestPerson** method that takes its input from the console:

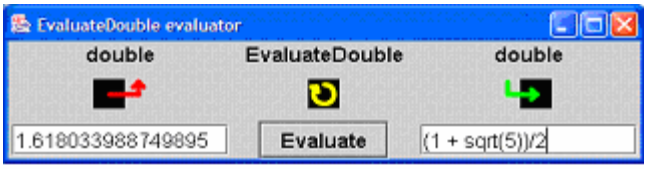
```
public static Person requestPerson()
    throws CancelledException
{ return new Person(
    console.in.requestString("Name:"),
    console.in.requestInt("Age:"));  }
```

Notice that we have built this method without even making the **Person** class **Stringable**. Indeed, as long as we are willing to create new objects when doing IO rather than modify existing objects, we can use static factory methods that mimic constructors as in the example above.

For the second example, consider the evaluation of a double precision number. Imagine the following proper method entered into a JPF **Methods** class:

```
public double EvaluateDouble(double x)
  { return x; }
```

At first glance, this method appears to do nothing interesting. Nevertheless, let's see what happens when this method is executed in the JPF GUI. Since the method has one argument and a return value, JPF will bring up an auxiliary frame as follows:



Notice that we have entered as the argument *(1 + sqrt(5))/2* in the input panel on the right and that after evaluation we have obtained the decimal value of this expression *1.618...* in the output panel on the left The reason this works is that in JPT all numeric input data is sent through the mathematical expression parser.

We conclude this section by noting that, since JPF automatically opens the console frame, console IO is immediately available in JPF and may be used for testing or other interaction. If a method in a JPF **Methods** class creates a GUI frame, then all of the IO facilities for GUIs are available for that frame.

4. GUI Composition

A central problem in GUI construction is the composition of the individual GUI widgets into some organized arrangement within a panel or frame. To avoid manual pixel positioning, Java uses layout managers to layout components. Unfortunately, the layout managers provided by Java are very problematic. The simple ones do not do enough and the advanced ones are very hard to use and often do not produce the expected result after all the work.

After some experimentation with layout managers, our colleague Jeff Raab invented **TableLayout**. This layout manager can handle either one or two dimensional layouts, always respects the preferred size of its components, permits any desired alignment of small components within table cells, and allows for user-defined spacing between cells. Once we began to use **TableLayout**, all of our problems with Java layout ceased.

Our next step in simplying GUI composition was to realize that we could make things easier by defining a panel **TablePanel** that must use **TableLayout** for its layout. We also realized that if we provided a rich set of constructors for **TablePanel** then the user could define entire sections of a GUI by a single constructor call. We further realized that if we did things right then **TablePanel**'s would nest in **TablePanel**'s recursively and entire frames could be constructed in minutes of work.

There are 4 families of constructors for **TablePanel** with a total of 19 constructors. Here is a description of each family:

- The simplest constructors supply structural data for the table but no content. The user must call *add* methods to add content to the table after construction. Because the table is not finished at the time of construction, these constructors are used in practice only rarely.
- The second family of constructors takes a 1-dimensional array of `Object` to provide content and an orientation (horizontal or vertical) that determines how the content of the table will be displayed.
- The third family of constructors takes a 2-dimensional array of `Object` to provide content. The visual display of the table parallels the structural layout of the array.
- The final family of constructors takes an algorithmic object that implements `TableGenerator`. Then, given the number of rows and columns in the table, this object will algorithmically construct the contents.

The constructors that use `Object` arrays are the ones that are used most often. An important aspect the design is that we accept a general `Object` rather than require a `Component`. This enables the user to think conceptually and to allow the details of `Component` construction to be encapsulated also.

The process of transforming an Object into a Component works as follows. A Component is inserted as is. A String or Icon is inserted into an Annotation object. A Shape is inserted into a ShapePaintable object and an Image into an ImagePaintable. In turn, the Paintable is inserted into a JPTComponent. An Action is inserted into a JButton. Finally, in all other cases, null is inserted which simply leaves a potential gap in the table.

The constructor family that uses a `TableGenerator` object is designed for those situations where the exact contents of the table cannot be known at compile time. This has pedagogical uses when building algorithm animations since what is being animated may depend on user choices at runtime. For example, if the user changes the number of items being animated, we can simply clear the table and then rebuild the table contents algorithmically.

The variety of ways to build a table is quite helpful to students and to faculty because it means that the right tool is available to do the desired task – it isn't necessary to use a screwdriver as a hammer.

In practice, GUI frames are built using nested `TablePanel`s. Once the main panel is built, we have a one line command to create its enclosing frame, pack it, and make it visible.

This rapid process of GUI composition that starts with arrays of objects, uses recursive nesting, and then creates the frame in one step has led us to introduce *eXtreme encapsulation*[ ] as a design idea. There is a qualitative difference in how an entity is built if one can create the entity as a whole rather than have to use a sequence of *add* methods to create the entity incrementally. Note also that this process exhibits a functional/declarative style that adds significantly to the clarity of the design.

5. GUI Widgets

Just as a tree structure requires leaf nodes, to create a GUI one must have entities to populate the tables. Popularly, such entities are called widgets. Often people think of widgets as simple but we will see that JPT supports both simple GUI widgets and widgets that bundle significant structure and functionality.

JPT can, of course, use any widget built into Java. In addition, JPT has a TextFieldView that integrates the robust input facilities into a GUI framework. JPT also provides a text area view, an enhanced check box called BooleanView, an enhanced slider called SliderView, and a ColorView that provides many options for setting color.

JPT supports that construction of widgets that use lists of strings, that is, radio button groups and dropdown lists. Following the same rapid construction philosophy as with tables, these widgets may be constructed in a single step using an appropriate string array. Let us discuss these widgets in more detail.

In a simple RadioPanel, the strings provided in the constructor are used to define the labels for the radio buttons in the panel. The program can query the selected button, button string, or button index and act accordingly. If desired, an action may be performed when a new button is selected by the user. In the extension of this class called StringObjectRadioPanel, the constructor is given a list of pairs consisting a string and an associated object in the model. The program can then directly query the *selected object* rather than needing to deal with buttons, strings, or indices.

For dropdown lists, there are two classes, DropdownView and StringObjectDropdown. A DropdownView has the same features as a TextFieldView except that the dropdown list provides a set of acceptable default choices rather than a single default choice. As an example, this view is used in the implementation of ColorView to permit color to be selected by name. A StringObjectDropdown uses a list of string-object pairs as in a StringObjectRadioPanel to permit the program to directly query the *selected object*. The key difference between DropdownView and StringObjectDropdown is that a DropdownView can permit user data entries just as in a TextFieldView whereas a StringObjectDropdown must use only the strings provided by the program since these are the only ones that pair with objects in the model.

6. Actions in JPT

From the earliest days of JPT, we wanted a simple way to attach behavior to elements of a GUI or to create such elements on the fly as desired. We decided to focus on the `Action` interface because this interface combines behavior via the *actionPerformed* method with a property list that may be used to reference button or menu text, icons, and tooltips. The `AbstractAction` class is the default implementation for `Action`.

One of our first steps was to build a factory method to construct a JButton from an Action. Later, in Java 1.3, Sun added a JButton constructor to do just this task.

From the point of view of pedagogy, however, there was one problem with the use of Action. The *actionPerformed* method takes an ActionEvent parameter and this parameter is almost never used for an Action associated with a button or menu item. Having to explain ActionEvent in order to define buttons is therefore a distraction.

We decided to define `SimpleAction` to avoid the ActionEvent parameter. This abstract class requires a simpler method:

public void perform();

The *actionPerformed* method is now a final method that calls *perform*. For convenience, we arranged that `SimpleAction` also implements ChangeListener and PropertyChangeListener by the same mechanism of calling *perform*.

We wanted SimpleAction to be used easily by students and this meant that we did not want students to have to define named classes that extend SimpleAction and implement *perform*. We therefore decided to use *anonymous inner classes* but under no circumstances were we ever going to use this scary terminology in the classroom. We decided instead to treat anonymous definition as a matter of syntax. We present the pattern:

```
SimpleAction action
  = new SimpleAction("action-name") {
      public void perform() {
        // action behavior here
      }
    };
```

We explain that when a constructor is called, Java permits the user to define necessary abstract methods by enclosing the method definitions in a pair of braces, { }, that immediately follow the parameter list. We do not say how Java implements this syntax. Later we explain that the same syntax may be used to redefine methods but that information is not needed at the start.

Now that it was easy to define the actions needed to create a set of buttons, we wanted to make it simple to both define the buttons and install them in a GUI in one step. We therefore introduced `ActionsPanel` whose constructor takes an array of actions and whose result is a panel containing the buttons constructed from the actions.

Historically, ActionsPanel predates TablePanel. Once we had invented TableLayout, it was natural to generalize ActionsPanel to TablePanel where an array of actions becomes an array of objects that are automatically converted into components.

There is one problem with the direct attachment of an action to a button. If the action is algorithmically intense then, while the algorithm is running, the GUI will be unreponsive, that is, the program will appear to hang. The solution is to run an intense algorithm in a separate thread but the question is how to do this simply.

Our solution is a wrapper class `ThreadedAction` that takes an action in its constructor and returns a new action. The new action sets up a new thread and causes the original action to be run in this new thread. This mechanism is so simple that it is easy to explain to students. Later, when synchronization is needed, that topic can be introduced and discussed. The *Kaleidoscope* case study available on the JPT web site provides an excellent example of how to deal with synchronization in a GUI environment.

There are a number of additional ways that actions are used in JPT and we will sketch these briefly. To handle the mouse and other event generators, we have defined action categories and adapters that enable users to attach one or more actions to particular kinds of events. For example, a MouseActionAdapter can accept actions to be executed for the following events: move, drag, press, release, click, enter, and exit. The action can either be a SimpleAction that does not need to know the mouse specifics or can be a MouseAction that requires the method

public void mouseActionPerformed(MouseEvent mevt)

and that will use the mouse event information in its execution. It turns out to be quite helpful to be able to mix simple actions and mouse actions together.

A final application of actions in JPT is GeneralDialog which replaces the cumbersome dialog classes in Java. A general dialog can be built with an arbitrary panel on top and a button panel on bottom that is generated by a set of dialog actions. A dialog action is a pair consisting of an ordinary action (that may interact with the panel) together with a specification of what should be done to the dialog when the action is complete: keep open, close, or cancel. With this generality, it is a pleasure to use dialogs rather than a hassle.

7. Graphics: BufferedPanel

The standard way to paint graphics on a Java component is to override its *paintComponent* method. This requires either that the graphics code is given inline or that there is some auxiliary data structure that provides the necessary data and instructions. This is not an easy model for beginning students to use.

We defined the panel `BufferedPanel` to permit painting to be done in steps with the program in control of when to refresh. The caller uses the method *getBufferGraphics* to obtain the graphics context of an internal Java BufferedImage. Graphics activity is therefore saved in the pixel color data of this internal buffer. When refresh is desired, the user calls *repaint()* or *quickRepaint()*. The *repaint()* method uses the standard Java mechanisms for repaint and may entail a slight delay. The *quickRepaint()* method forces immediate repaint and is useful for animation.

The Java Power Framework provides a 400 x 400 BufferedPanel for its graphics window. Since a BufferedPanel will accumulate whatever is painted onto it (until the next clear operation), it is possible to conduct a series of graphics experiments and to see the results side-by-side. This enables decisions about what to paint to be made by students at runtime rather than at compile time.

A BufferedPanel comes with a built-in MouseActionAdapter. Using this adapter, one can do simple mouse-driven graphics very easily. When it is time for more sophisticated mouse-driven graphics, the student may use Paintable and MutatablePaintable objects that will be described in the next section.

8. Graphics: Paintable & MutatablePaintable

The `Paintable` and `MutatablePaintable` interfaces were invented to:

- Allow shapes, images, and text to be treated as graphics entities in a uniform fashion.
- Enable the mutation of these graphics entities by affine transforms without the need to know internal details of the entities.
- Support the *composite pattern* so that a sequence of paintables could also be treated as a paintable.

The most important method required by Paintable is:

void paint(Graphics g).

This method may be called in the *paintComponent* method for a component or can be used directly to paint onto a BufferedPanel. A base definition of this method is given in the specific Paintable objects defined for shapes, images, and text. The definition in composite structures recursively reduces to these base definitions.

Paintable also requires method to get the bounds and center of the object, to determine if a point is within the paintable region, to set visibility, and to set opacity (or transparency if you will).

A MutatablePaintable is a Paintable that manages an invertible affine transform that is applied to the graphics context before the paint operation and then undone afterwards. In fact, Java does not permit distortion of a graphics context by a transform that is not invertible. Because distortion is applied to the graphics context rather than to the object being rendered, there is no need to know the structural details of the object or even its type.

The wrapper class MutatableWrapper will wrap a Paintable with an affine transform and make it MutatablePaintable. The class PaintableSequence goes even further. This class accepts an array of Paintables in its constructor. As it installs each Paintable, it checks if it is MutatablePaintable and, if it is not, then it wraps the Paintable using a MutatableWrapper. The PaintableSequence also maintains its own mutator so that it may be mutated as a whole. This design means that one can algorithmically build sequences of Paintables with arbitrary levels of nested sequences.

The base classes in this structure that deal with shapes, images, and text also provide flexible functionality. As an example, consider shapes. When building a ShapePaintable from a Shape, one can optionally supply a paint mode (fill, draw, or fill-draw), a Paint for fill, a Paint for drawing the boundary, and a Stroke for specifying how the boundary is drawn. If parameters are omitted, sensible defaults are used.

In our opinion, once students have learned how to paint or draw the simple built-in Java shapes (rectangles, ellipses, lines, ...), the Paintable and MutatablePaintable concepts provide a rich path forward into more advanced graphics that truly demonstrates the power of object-oriented design principles to integrate disparate entities (shapes, images, and text) into a common framework. Of course, since JPT is 100% open source, students can read how the classes are designed as soon as they have sufficient maturity.

9. Graphics: Shape Generation

As we were dealing with ShapePaintable objects, we realized that Java once again provides the atomic tools for shape construction but does not provide an integrated mechanism to make building curved shapes easy. What Java provides for curves are the tools to construct a quadratic or cubic Bézier arch. If you want to join such arches to make a larger curve, you must call methods do this yourself in a one-by-one process.

We decided to attempt to solve the general problem of building shapes given a set of vertices and, optionally, a set of tangents at these vertices. There are immediately two options: *automatic* (compute tangents from vertices algorithmically) and *tweakable* (specify both vertices and tangents). We therefore decided on two classes AutomaticShape and TweakableShape each derived from a common class BaseShape.

For the *automatic* case, we wanted to supply two algorithms for computing tangents from vertices but we also wanted to leave the door open for users to supply other algorithms. We therefore defined an interface Tangent.Strategy that such algorithms would need to meet. We then introduced two strategies, bezierStrategy and chordStrategy. The bezierStrategy implements an algorithm for the unique piecewise cubic curve that passes through the given vertices, has the given tangents, and is twice differentiable[38]. The chordStrategy is much simpler: it just computes the tangent at a vertex as a fixed multiple of the adjacent chord. There are other strategies in the graphics literature but we did not implement them in JPT.

The common class BaseShape provides some useful options. As we were testing curve drawing, we found that we wanted to see the curve itself, the internal polygon joining the vertices, the external polygon that includes the vertices and the tangents, the vertex dots alone, and the tangent segments alone. We decided to provide a parameter in BaseShape to select which of these options would be drawn. In particular, it is therefore a simple toggle to switch between a smooth curve and a polygon. The other BaseShape options control whether the curved is open or closed and which Java area fill algorithm is used for fill.

We also decided to make public our testing tools for shapes so that students could use the tools not only to draw shapes but to investigate how the shapes are rendered. Such an investigation is facilitated by the following static method in class Path that does everything: fills the shape, draws the outline, draws the Bézier frame, and draws the vertex and tangent dots.

Path.showShapeStructure(...)

For example, it is quite interesting for students to learn from this method how Java renders a circle (as 4 Bézier cubic arches).

So far, we have described how to systematically create a smooth cubic curve or the associated polygon given an array of vertices and, optionally, an array of tangents. What about curves joined with sharp corners or even with gaps that are left unconnected? We handle this by *Path.append* that takes an array of Shape objects and a boolean array that determines whether to connect one Shape to the next or not and returns a Java GeneralPath that also implements the Shape interface.

In summary, we use arrays of vertices and tangents to build smooth shape objects or polygons and then use arrays of shapes to build broken or disjoint shape objects. Once again, the principles of *eXtreme encapsulation* lead to an elegant solution to a complex problem.

10. Graphics: Plot Tools

The JPT has tools to plot one or more numerical data sets as if these sets represented mathematical functions. The tools will automatically scale the world coordinates of the data to the image coordinates of a desired screen rectangle. The tools can also draw axes, grids, and tick marks.

11. The Quick Classes

Many constructors in JPT use arrays for quick initialization of objects. In the latest release of JPT, we decided to apply the same idea to the Java collection classes. We therefore extended the following 8 classes to add constructors that utilize arrays: Vector, ArrayList, LinkedList, HashSet, TreeSet, Hashtable, HashMap, and TreeMap. We also provide access to the structures that manage string-object pairs for the string-object GUI views.

12. Comments and Conclusions

The Java Power Tools is a large, powerful library of classes that is nevertheless very simple to use. A great deal of design effort has gone into making these tools both powerful and simple. In addition, we have been mindful of the practical experience of teaching students and designing labs and demos. If we wish to have students do something and it is not as simple to do using pure Java as we feel it should be then we add methods or classes to JPT to provide support. If we find something in JPT that is awkward to use, we ruthlessly refactor. We want students to have a good experience with JPT and we do not want them to be disappointed if they look under the hood at its code.

Given the care we have taken with JPT and the fact that it is 100% open source, students can fruitfully use JPT after the freshmen year. Indeed, juniors and seniors have used JPT in their upper level projects to speed GUI development.

We would certainly recommend JPT to anyone who wants to teach Java using robust IO and well defined GUIs. We would also recommend JPT even if the only tool being used is the Java Power Framework. The JPF provides students with the power to freely explore and experiment and we would hope that all faculty would see this as pedagogically valuable.

There are some faculty who believe that students should only be taught concepts and APIs that are in the "official language". Once before we addressed this topic[39], but it is worth saying a few remarks here about this issue here.

We believe that it is not the job of university education to provide specific job training for the marketplace. Vendor tools come and go and our role has to be to use ideas and tools that have some lasting conceptual value. Eleven years ago, Eric Roberts[45] faced the criticism that his C tools did not follow ANSI C.

> "One reader felt sufficiently strongly about this question to set an entire sentence in italics: '*If you're going to teach ANSI C, teach ANSI C, not some local modified version of the language*'.
>
> "I believe that the last comment cuts to the heart of the controversy. I was not trying to teach ANSI C. I was trying to teach programming."

Roberts then continued with a summary of the principles he was trying to teach at the time and concluded:

> "The fact that our course follows its own advice with respect to library development encourages students to do the same."

The Java Power Tools has been intended from the start to be not only a useful set of tools for freshmen courses but also a library that could be used in later courses and one that could withstand close inspection in courses on object-oriented design. These tools are built both to work smoothly and to be read.

We suspect that some faculty do not wish to use tools because they fear that students will not be prepared for the job market. For us that seems really odd because Northeastern is noted for its cooperative education program and our students enter the job market on "coop" much sooner than students from other schools. Our view is that their exposure to tools is one factor that helps them succeed because it gives them a more abstract perspective and also teaches them that you do not have to accept a vendor package as is.

Moreover, if a person can develop GUIs using JPT in 1/10 of the time it takes using pure Java, then this shows students what is possible and does not limit them to the current state-of-the-art. Indeed, it may cause them to ask why the state-of-the-art is not in fact much better than it actually is.

The deepest question for faculty though is this: What is education all about? We do not believe that education is simply about teaching *what is*. We believe that education is about dreams, vision, and *what might be*.

13. Guide to the References

Because the ACM Task Force[51, 52] is currently examining Java and Java APIs for use in pedagogy, we have listed many of the key papers on this subject over the last several years.

Our own work on Java Power Tools is discussed in [28, 32, 33, 34, 35, 36, 37, 40, 41, 42].

Selected pointers to the work of others on pedagogical APIs for Java or other languages are [4, 5, 7, 8, 9, 11, 12, 13, 14, 15, 17, 23, 24, 25, 26, 29, 30, 31, 43, 53, 54].

Selected pointers to pedagogical IDEs and related tools are [1, 10, 16, 20, 21, 44].

Selected pointers to more general work on object-oriented design and eXtreme programming are [2, 3, 18, 19, 22].

Finally, selected pointers to the work of Eric Roberts which has always been one of our inspirations are [45, 46, 47, 48, 49, 50, 51, 52].

References

[1] Barnes, D., and Kölling, M., *Objects First With Java: A Practical Introduction Using BlueJ*, Prentice-Hall, 2003.

[2] Beck, K., *Extreme Programming Explained*, Addison-Wesley, 2000.

[3] Beck, K., *Test Driven Development By Example*, Addison-Wesley, 2003.

[4] Bergin, J., Naps, T. L, et al, *Java Resources for Computer Science Instruction*, SIGCSE Bulletin, 30(4), 1998, 14-24.

[5] Bergin J, Proulx, V. K., et al, *Resources for Next Generation Introductory CS courses*, SIGCSE Bulletin, 31(4), 1999, 101-105.

[6] Berners-Lee, T., *Weaving The Web*, Harper, 1999 (see 40).

[7] Bishop, J., and Bishop, N., *Object-orientation in Java for scientific programmers*, SIGCSE Bulletin, 32(1), 2000, 357-361.

[8] Bishop, J., and Horspool, N., *Developing Principles of GUI Programming Using Views*, SIGCSE Bulletin, 36(1), 2004, 373-377.

[9] Bishop, J., and Horspool, N., *C# Concisely*, Addison-Wesley, 2004.

[10] BlueJ: http://www.bluej.org/.

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

[12] Bruce, K., Danyluk, A., and Murtagh, T., *Event-driven programming is simple enough for CS1*, SIGCSE Bulletin, 33(3), 2001, 1-4.

[13] Caspersen, M., and Christensen, H., *Here, There and Everywhere - On the Recurring Use of Turtle Graphics in CS1*, SIGCSE Australasian Conference, 2000, 34-40.

[14] Christensen, H., and Caspersen, M., *Frameworks in CS1: A Different Way of Introducing Event-Driven Programming*, SIGCSE Bulletin, 34(3), 2002, 75-79.

[15] Cooper, S., Dann, W., and Pausch, R., *Teaching Objects-First in Introductory Computer Science*, SIGCSE Bulletin, 335(1), 2003, 191-195.

[16] Eclipse: http://www.eclipse.org/.

[17] Ellis, A., Hagan, D., et al, *A Collaborative Strategy for Developing Shared Java Teaching Resources to Support First Year Programming*, SIGCSE Bulletin, 31(3), 1999, 84-87.

[18] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[19] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[20] jGRASP: http://jgrasp.org.

[21] JUnit: http://www.junit.org/.

[22] Kay, A. C., *The Early History of Smalltalk*, in Bergin, T. J, & Gibson, R. G., *History of Programming Languages – II*, ACM Press, 1996, 511-598 (see 542-543).

[23] Koffman, E. and Wolz, U., *CS1 using Java Language Features Gently*, SIGCSE Bulletin, 31(3), 1999, 40-43.

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

[25] Koffman, E. and Wolz, U., *Problem Solving with Java*, Addison-Wesley, 2001.

[26] Lambert, K. and Osborne, M., *Easy, realistic GUIs with Java in CS1*, The Journal for Computing in Small Colleges, 16(2), 2001, 209-215.

[27] Marchesi, M., Succi, G., Wells, D., and Williams, L., *Extreme Programming Perspectives*, Addison-Wesley, 2003.

[28] McDonald, J., *Interactive Pushdown Automata Animation*, SIGCSE Bulletin, 34(1), 2002, 376-380.

[29] Mitchell, W., *A Paradigm Shift to OOP has Occurred…Implementation to Follow*, The Journal for Computing in Small Colleges, 16(2), 2001, 94-105.

[30] Nevison, C., and Wells, B., *Teaching Objects Eearly and Design Patterns in Java using Case Studies*, SIGCSE Bulletin, 35(3), 2003, 94-98.

[31] Parlante, N., *Teaching with Object Oriented Libraries*, SIGCSE Bulletin, 29(1), 1997, 140-144.

[32] Proulx, V. K., *Programming Patterns and Design Patterns in the Introductory Computer Science Course*, SIGCSE Bulletin, 32(1), 2000, 80-84.

[33] Proulx, V. K., Raab, J., and Rasala, R., *Traffic Light: A Pedagogical Exploration Through a Design Space*, The Journal for Computing in Small Colleges, 15(5), 2000, 25-31.

[34] Proulx, V. K., Rasala, R., and Rodrigues, J. J., *Simple Problem Solving in Java: A Problem Set Framework*, The Journal for Computing in Small Colleges, 17(6), 2002, 56-70.

[35] Proulx, V. K., Raab, J., and Rasala, R., *Objects from the Beginning - with GUIs*, SIGCSE Bulletin, 34(3), 2002, 65-69.

[36] Proulx, V. K. and Rasala, R., *Java IO and Testing Made Simple*, SIGCSE Bulletin, 36(1), 2004, 161-165.

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

[38] Rasala, R., *Explicit Cubic Spline Interpolation Formulas*, in Glassner, A. S., *Graphics Gems*, Academic Press, 1990, 579-584.

[39] Rasala, R., *Toolkits in First Year Computer Science: A Pedagogical Imperative*, SIGCSE Bulletin, 32(1), 2000, 185-191.

[40] 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.

[41] Rasala, R., Raab J., and Proulx, V. K., *The SIGCSE 2001 Maze Demonstration Program*, SIGCSE Bulletin, 34(1), 2002, 287-291.

[42] Rasala, R., *Embryonic Object versus Mature Object: Object-Oriented Style and Pedagogical Theme*, SIGCSE Bulletin, 35(3), 2003, 89-93.

[43] Reges, S., *Conservatively Radical Java in CS1*, SIGCSE Bulletin, 32(1), 2000, 85-89.

[44] Reis, C., and Cartwright, R., *Taming a Professional IDE for the Classroom*, SIGCSE Bulletin, 36(1), 2004, 156-160.

[45] Roberts, E., *Using C in CS1: Evaluating the Stanford Experience*, SIGCSE Bulletin, 25(1), 1993, 117-121.

[46] Roberts, E., *The Art and Science of C: A Library-Based Introduction to Computer Science*, Addison-Wesley, 1994.

[47] Roberts, E., A C-based graphics library for CS1, SIGCSE Bulletin, 27(1), 1995, 163-167.

[48] Roberts, E., *Programming Abstractions in C: A Second Course in Computer Science*, Addison-Wesley, 1997.

[49] Roberts, E. and Picard, A., *Designing a Java Graphics Library for CS 1*, SIGCSE Bulletin, 30(3), 1998, 213-218.

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

[51] Roberts, E., *The Dream of a Common Language: The Search for Simplicity and Stability in Computer Science Education*, SIGCSE Bulletin, 36(1), 2004, 115-119.

[52] Roberts, E., *Resources to Support the Use of Java in Introductory Computer Science*, SIGCSE Bulletin, 36(1), 2004, 233-234.

[53] Wolz, U. and Koffman, E., *simpleIO: A Java Package for Novice Interactive and Graphics Programming*, SIGCSE Bulletin, 31(3), 1999, 139-142.

[54] Woodworth, P., and Dann, W., *Integrating Console and Event-Driven Models in CS1*, SIGCSE Bulletin, 31(1), 1999, 132-135.