Java Power Tools:
Model Software for Teaching Object-Oriented Design

Richard Rasala, Jeff Raab, Viera K. Proulx
College of Computer Science
Northeastern University
Boston MA 02115
{rasala,jmr,vkp}@ccs.neu.edu

Abstract

The *Java Power Tools* or JPT is a Java toolkit designed to enable students to rapidly develop graphical user interfaces in freshman computer science programming projects. Because it is simple to create GUIs using JPT, students can focus on the more fundamental issues of computer science rather than on widget management. In a separate article[4], we will discuss with examples how the JPT can help freshman students to learn about the basics of algorithms, data structures, classes, and interface design. In this article, we will focus on how the JPT itself can be used as an extended case study of object-oriented design principles in a more advanced course.

The fundamental design principles of the JPT are that *the elements of a graphical user interface should be able to be combined recursively as nested views* and that *the communication between these views and the internal data models should be as automatic as possible*. In particular, in JPT, the totality of user input from a complex view can be easily converted into a corresponding data model and any input errors will be detected and corrected along the way. This ease of communication is achieved by using string objects as a lingua franca for views and models and by using parsing when appropriate to automatically check for errors and trigger recovery. The JPT achieves its power by a combination of computer science and software design principles. Recursion, abstraction, and encapsulation are systematically used to create GUI tools of great flexibility. It should be noted that a much simpler pedagogical package for Java IO was recently presented in [9].

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

SIGCSE 2001.

1. The MVC Paradigm and the JPT

The Java Power Tools are based on the well-known model-view-controller paradigm. In this paradigm, the internal data models are separate from the interface views that set or display the data. This design allows the models to be represented by a variety of views as desired. In addition, in the MVC paradigm, the algorithmic logic of the model is separate from the widget management of the view thus creating a much simpler program design.

In order for MVC design to work, the views must maintain their own state and there must be communication in which the models update the views or the views update the models. This communication may be initiated by internal processes or by the user activation of controls. The controls may also trigger the internal processes that lead to an update of both the models and the views.

Although the model-view-controller paradigm provides a general framework for interface design, there are several issues that remain problematic:

- The communication mechanism between the models and the views is not specified and may vary from view to view. This means that communication does not scale automatically to more complicated views.
- The relation between the controls and the model-view communication is also not specified and must often be arranged manually for each model-view pair.
- Views do not have a common data representation that will facilitate communication.
- The model-view-controller paradigm has a missing piece which may be illustrated by the following table:

Internal Element	Interface Element
Model	View
?	Control

To mirror controls on the internal side there needs to be an encapsulation of the corresponding actions.

- The model-view-controller paradigm does not specify an error handling strategy.

In the current implementation of Java using the elegant Swing components, the above problems are still present. Although the Swing widgets provide a starting point for interface design and implementation, much repetitive work must be done to craft a working program. In many books on Java, the same code sequences occur over and over. This is a sure sign that fundamental abstractions and encapsulations have not been recognized.

The design goal of the Java Power Tools is to articulate these abstractions and encapsulations so that building a graphical user interface can be made smooth and easy. Furthermore, because building user interfaces is a central issue in modern software development, we believe that the problems addressed by the JPT and the techniques used for their solution form an excellent case study in a course on methods of object-oriented design.

2. Fundamental Design Features of the JPT

The Java Power Tools actually support three levels of user interface design:

- A traditional console model for input-output.
- Simple dialog boxes for input of basic types.
- Full graphical user interfaces that are built from basic view components using recursive concatenation and nesting and that adhere to the MVC paradigm.

To maintain an economy of code, to support all three levels of interface design, to solve the problems with the MVC paradigm, and to remain as close as possible to pure Java, the JPT meets the following design constraints:

- The input of basic types must use a common parsing technology and error detection strategy that applies to all three levels of user interface design.
- User interface views must be extensible by the operations of concatenation and nesting.
- Views must be displayable either within frames or in dialog boxes.
- Model-view communication must scale as views of greater complexity are built.
- The error correction strategy for graphical interfaces must itself be graphical.
- User interface controls must be easy to install and connect with internal actions.
- Whenever possible, JPT objects must extend Java objects so that no Java functionality is obscured or lost by using JPT.

We will now describe the JPT design in detail.

2.1 Strings: The Key to Model-View Communication

In Java, different interface widgets deliver different types of data. For example, a JTextField produces a String while a JSlider produces an integer in a bounded range. Such type differences tend to work against a recursive model for building graphical user interfaces since it is difficult to deal directly with the multiple types in a uniform manner. Perhaps the most critical design decision in all of the JPT is the realization that to support a recursive paradigm all views must be able to transmit data information using a single type, namely, String.

The type String is fundamental for several reasons. All of the primitive data types may be represented using strings. Similarly, the essential state information in primitive views may also be represented using strings. This means that more complicated models and views that are created by concatenation and nesting may be represented by suitable combinations of strings.

The key point is that we have 1-to-1 maps of models to strings and of views to strings. This is the basis for the model-view communication. If we have a model, we compute its string and then use this string to set the state of the corresponding view. In the reverse direction, if we have a view, we compute its string and then attempt to use the string to set the data in the model. In this direction, input errors may occur since the user may have improperly set the view state. These errors will be detected during the parsing operations that set the data in the model and appropriate recovery actions will be invoked by JPT automatically.

Our decision to use structured String objects to capture the essence of model-view communication is similar to the decision made by Tim Berners-Lee [1] in his choice of a structured text based language, HTML, as a foundation for web communication:

> "I expected all kinds of data formats to exist on the Web. I also felt there had to be one common lingua franca that any computer would be required to understand."

The use of structured text as an extensible mechanism to enable communication of diverse kinds of information is a design technique that should be highlighted in courses on software design.

2.2 The Stringable Interface

The Stringable interface defines the required behavior of data model objects whose state may be encapsulated into a String. This interface has just two methods:

```
String toData()
void fromData(String data)
```

The toData method encapsulates the state of the model into a String. The fromData method reverses this process by attempting to set the state of the model from the data String. If fromData cannot set the state of the model from

the string then a ParseException is thrown. This exception will set in motion the error handling strategies.

Because the fromData method changes the state of a Stringable object, a Stringable object must be mutable. This means that String itself cannot be made Stringable and that the Java object wrappers for the primitive types also cannot be made Stringable. This is unfortunate.

With some reluctance, we made the decision to define 9 Stringable base types corresponding to the String class and to the 8 primitive Java types. We call these types XString, XBoolean, ..., where the X stands for "extra". These X-types form the base objects for building more general Stringable objects by recursion.

The X-types incorporate sophisticated parsing within the fromData method so that arithmetic/boolean expressions may be evaluated as easily as pure constants. This means that expression evaluation is available for input of the primitive types in all contexts: console or graphical.

To use the X-types recursively to make more complicated Stringable data models, we have codecs (encoders and decoders) that bundle and unbundle concatenated String objects without loss of information.

The result of these design decisions and features is that it is easy to create Stringable data models of *arbitrary complexity*.

2.3 The Displayable Interface

The Displayable interface defines the required behavior of input objects that will be displayed on the screen. This interface has five methods:

```
String getViewState()
void setViewState(String data)
void setDefaultViewState(String data)
void reset()
void setEnabled(boolean setting)
```

The getViewState method encapsulates the state of the view into a String. The setViewState method reverses this process and sets the state of the view from the data String. The setDefaultViewState method sets the default contents of the view for the reset method. The setEnabled method recursively enables or disables nested views.

The most important class that implements the Displayable interface is the DisplayPanel. This class extends the Java class JPanel so that all features of standard panels may be used including layout managers and borders. The DisplayPanel class is a container class that is intended to hold Java components. The active components that are added to a DisplayPanel should be Displayable. The inert components (labels, icons) may be arbitrary.

The DisplayPanel is fundamental because it implements the recursive strategy for building GUIs. The view state of a DisplayPanel is defined as the concatenation of the view states of its Displayable components using the current codec of the panel for encoding-decoding. This means that we have the basic tool for aggregating many smaller view objects into one large view object.

There are three classes that extend DisplayPanel and provide special services: Display, DisplayCollection, and ArrayPanel. The Display class encapsulates a view together with an optional annotation (text and/or icon) and an optional titled border. The DisplayCollection class specializes the DisplayPanel class by restricting the layout to horizontal or vertical. The ArrayPanel class focuses on the display of a dynamic array whose size may be changed by either the user or the program. This class supports automatic scrolling as needed.

The various classes that directly implement Displayable provide the recursive tools for building GUI's. We will next focus on some of the ingredients for the GUI's.

2.4 Displays with Type Information

If we have a Stringable model M and a Displayable view V then we can use V to set M via the code:

```
M.fromData(V.getViewState());
```

Of course, we need to deal with a possible ParseException that may be thrown by fromData. This means wrapping the above line of code in a try...catch... clause and dealing with the potential error using dialog boxes. This code sequence follows a pattern and therefore begs to be encapsulated. This key to this encapsulation is to have V know the class of the object M that it must define or update.

The TypedView interface extends the Displayable interface with the following five methods:

```
Class getDataType()
Stringable demandObject()
Stringable requestObject()
InputProperties getInputProperties()
void setInputProperties(InputProperties p)
```

If V is a TypedView then it has an associated data type that represents the type of object it can provide input for. This is the class returned by the method getDataType. If this class inherits from the data type T of M then we can use the following code to perform the input operation:

```
M = (T) V.demandObject();
```

or

```
M = (T) V.requestObject();
```

Notice that rather than modify the current object M we now use a hidden factory method to create a replacement object that is assigned to M via the method calls.

The two methods, demandObject and requestObject, both handle input errors detected in the view but correspond to different input paradigms. When using demandObject, the method call must succeed and a valid data object must be stored in M. When using requestObject, we permit the user to cancel the input operation if an error is detected. In this situation, requestObject will throw an exception to notify

the caller so that any action that depends on successful input must be halted.

The InputProperties for a TypedView maintain a property list that records certain information needed if the TypedView is placed in a dialog box.

Some views such as text fields are capable of providing the state information for more than one type of object. In this case it is critical that the type of object be a settable parameter. This leads to the GeneralView interface which extends TypedView by adding one additional method to permit the class of the associated model object to be set:

```
void setDataType(Class dataType)
```

2.5 Basic Views

We will now summarize the basic views that are currently available in the Java Power Tools.

The most important view is the TextFieldView which is a GeneralView directly capable of obtaining the String data needed to initialize String objects or any of the 8 primitive types. Technically, a TextFieldView produces one of the nine X-types discussed above. Static methods allow direct conversion to the built-in types. In theory, it is possible to use a TextFieldView for a more general Stringable object. However, it is easier to use multiple annotated text fields than to ask the user to encode data in a single text field.

The TextAreaView is used for multiples lines of text that may be automatically wrapped. Since it is not clear what the application program will do with this text, the text in this view is simply returned as an XString object.

The views that support interface widgets are as follows:

View	Widget
BooleanView	Check box
OptionsView	Radio button group
DropdownView	Dropdown text list
ColorView	Color sample & chooser
SliderView	Slider

2.6 Controls and Encapsulation of Actions

Just as we wish to encapsulate the creation of models and views and their communication, we also wish to make the creation of controls and the specification of their actions as simple as possible. A critical design technique is to make actions into objects using the AbstractAction class defined by Java.

Tha AbstractAction class provides in its constructors the ability to set a name and an icon for the action object. All that needs to be done is to instantiate the actionPerformed method that is initially abstract. This best way to do this is to define the action object within a class in such a way that the actionPerformed method calls a member function of the class as its only task. Then the action can take full advantage of the state of the enclosing class and all of its methods.

Frequently in user interfaces, actions are attached to buttons. To streamline the creation of buttons and their associated actions, the JPT introduces an ActionsPanel class that accepts actions. When an action is fed to an ActionsPanel, a button with the name or icon of the action is created automatically and a button listener is established that will perform the action when the button is pressed. In general, the JPT philosophy is to use various kinds of actions to hide the widget details as much as possible.

2.7 Dialogs and Error Handling

The JPT has three levels of dialog box: JPTDialog, InputDialog, and ErrorDialog. The JPTDialog class is a general class for building dialogs. Its constructor requires a TypedView that handles the user input. The InputDialog class extends JPTDialog to provide a modal input dialog. The internal TypedView is used to obtain the user data. This class predefines four actions that manage the input process: OK, Reset, Suggest, and Cancel. The ErrorDialog class extends InputDialog and makes certain adjustments appropriate to error handling. In general, a TypedView is responsible for its error handling and will create an ErrorDialog if an error is discovered by its input methods. This encapsulated code manages the error resolution.

3. Pedagogical Applications of the JPT

In the practical world, object-oriented programming is most valuable in large projects that make extensive use of libraries that are adapted and extended. One pedagogical problem in teaching OOP is that the small-scale textbook examples are not of a size that convinces a student that OOP is worth all of the fuss. Until a student experiences a large project in which the value of OOP is evident there is skepticism that the complications of defining classes and methods are valuable.

The pedagogical value of the JPT derives from the fact that the tools demonstrate the power of OOP by dramatically simplifying the creation of GUI's. A comparison of programs written using the Java Power Tools with those written in pure Java shows that there are enormous reductions in both the size and the complexity of the code. This gives a student strong motivation to ask the question: How is this power achieved?

The one sentence answer to this question is that the Java Power Tools utilize fundamental principles of computer science and object-oriented design to enable GUI software to be developed quite rapidly. Using the Java Power Tools as a case study, one can enumerate these principles and illustrate them with many examples. Here are the most important ones:

- *Recursion*: Recursion is a technique that is honored in the field of algorithms and is a way of life for the LISP

and Scheme communities. It is less well known that the same principles underlie the use of container classes in data structures and that these methods can be adapted to building GUI's. The JPT shows that if you think of the ingredients of a GUI in a recursive manner then you can achieve great power. The simple views are the base objects for recursion and the Displayable container objects are the means for recursive growth.

- *Abstraction*: The use of Java interfaces in combination with traditional inheritance illustrates how abstraction is maximally leveraged in an object-oriented context. For example, all of the views in JPT inherit state and behavior from different Java Swing widgets but become joined together by implementing common interfaces.
- *Encapsulation*: Encapsulation is the means by which we organize our thinking and avoid the dangerous practice of code repetition. In creating the JPT, any time we observed a common code sequence, it was abstracted and encapsulated. For example, the process of adding several views to a panel is encapsulated in DisplayCollection and the process of dynamically changing the number of views of a given type is encapsulated in ArrayPanel.
- *Actions*: The model-view-controller paradigm should really be called the *model-view-action-controller paradigm* since it as important to abstract and encapsulate actions as it is to organize the other three GUI ingredients. In the JPT, when we feed an action to an ActionsPanel, we encapsulate the entire process of defining a button, locating it in the panel, and defining a listener that will respond to the button press by performing the desired action.
- *Factories*: Object factories are used in a number of places in the JPT. The most notable example is the StringableFactory which can build a Stringable object given its Java class and the encoded String that defines its state.

In general, many specific object-oriented design patterns [3] may be found by looking at the JPT.

Finally, we should emphasize the code quality of the JPT. From the very beginning, we planned that the JPT would be used not only to enable freshman education but also to illustrate object-oriented design for upper level courses. All code is fully provided with Java docs. Painstaking care has been taken with the naming of classes, methods, and variables. The code has been formatted so that it can be displayed in a classroom at 16 point font for easy viewing by students. Finally, after one year of development, the team spent two months on code review and refactoring [2] so that the quality of the code and its abstractions would be the best that we could deliver. We believe that there is probably no code on the planet of its size (*300 pages*) that is as well suited for a case study as the Java Power Tools.

4. Conclusions

As educators, we have spent more than ten years in designing pedagogical toolkits for Pascal, C++, and now Java [4-8]. We firmly believe that toolkits are essential for pedagogy since students must understand and learn to practice the three grand themes of computer science: *algorithmics, encapsulation, and interaction*.

In the older traditional world of procedural programming, it was enough for computer science to focus on the algorithms because these were the invariant components of software. Now, however, since object-oriented thinking has placed a premium on abstraction and encapsulation and since interaction is a key aspect of all computing, it is critical that we start students on the road to learning the three grand themes as early as possible. When it comes time to explore design issues in the upper level courses, we must provide students with case studies that are serious, elegant, powerful, and compelling. We believe that the Java Power Tools meet this standard of excellence.

The Java Power Tools and related sample files may be found at:

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

References

[1] Berners-Lee, Tim, with Fischetti, Mark, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*, Harper San Francisco, 1999, 40.

[2] Fowler, Martin, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Reading MA, 1999.

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

[4] Proulx, Viera K., Raab, Jeff, and Rasala, Richard, *Building Java Graphical User Interfaces in Introductory Programming Courses,* in preparation.

[5] Raab, Jeff, Rasala, Richard, and Proulx, Viera K., *Pedagogical Power Tools for Teaching Java*, SIGCSE Bulletin, 32(3), 2000, 156-159.

[6] Rasala, Richard, *Design Issues in Computer Science Education*, SIGCSE Bulletin, 29(4), 1997, 4-7.

[7] Rasala, Richard, *Functions Objects, Function Templates, and Passage by Behavior*, SIGCSE Bulletin, 29(1), 1997, 35-38.

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

[9] Wolz, Ursula, and Koffman, Elliot, simpleIO: *A Java package for Novice Interactive and Graphics Programming*, SIGCSE Bulletin, 31(3), 1999, 139-142.