A Demonstration of JPie: An Environment for Live Software Construction in Java

Kenneth J. Goldman
Computer Science and Engineering
Washington University
St. Louis, Missouri USA
(314) 935-7542
kjg@cse.wustl.edu

jpie@cse.wustl.edu

ABSTRACT

JPie is a tightly integrated development environment supporting live object-oriented software construction in Java. JPie embodies the notion of a *dynamic class* whose signature and implementation can be modified at run time, with changes taking effect immediately upon existing instances of the class. The result is complete elimination of the edit-compile-test cycle. JPie users create and modify class definitions through direct manipulation of visual representations of program abstractions. This support is provided without modification of the language or run-time system. In this demonstration, we illustrate central features of JPie through the construction of a sample application. These include dynamic declaration of instance variables and methods, dynamic modification of method bodies and threads, dynamic user interface construction and event handling, and on-the-fly exception handling in JPie's integrated thread-oriented debugger.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**] Programming Environments – *graphical environments, integrated environments, interactive environments.* D.1.5 and D.1.7 [**Programming Techniques**] Object-oriented Programming and Visual Programming.

General Terms: Design, Human Factors, Languages.

Keywords: Live programming, dynamic classes, tightly integrated development environments, direct manipulation.

1 INTRODUCTION

This demonstration introduces JPie, a unique programming environment designed to make the power of object-oriented software development accessible to a wider audience. To accomplish this, JPie supports live software development through direct manipulation of graphical representations of programming language abstractions.

JPie is not a new programming language, but rather a tightly integrated approach to programming environments in which the environment is intimately aware of the structure of the software being developed, and therefore can support a high level of interactivity while harnessing the power of an underlying modern type-safe object-oriented language (Java).

Screen shots from the demo are available on our web site. [1]

1.1 Dynamic Classes

JPie's interactivity rests on the notion of a *dynamic class,* whose signature and implementation can be modified live, even while instances of that class exist in a running program. Dynamic classes fully interoperate with compiled classes. Consequently, JPie users have available the entire Java API (1.4), may create dynamic classes extending either dynamic or compiled classes, and can override methods on the fly. Instances of compiled classes may hold type-safe references to instances of dynamic classes, and may call methods on them polymorphically.

Dynamic classes are *precompiled* so that they may present themselves to the Java Virtual Machine (JVM) just as any ordinary type. However, they execute in a *semi-interpreted* manner using an internal representation of the dynamic portions of the class definition. The precompiled class overrides all of its inherited methods, and by default calls the parent method to carry out the computation. However, when the JPie user dynamically overrides a method, the user's implementation, rather than the parent's, is invoked. These capabilities rely extensively on Java's reflection mechanism and are accomplished without modification of the language or JVM.

1.2 Direct Manipulation

To make software development immediate and tangible, JPie provides graphical representations of programming language abstractions. For example, variables are shown as capsules whose type and scope are represented visually. JPie programmers manipulate the graphical representations to effect changes in the running program. Many operations, such as variable and method declaration and use, are accomplished by drag-and-drop. The graphical representations expose the Java execution model, while the JPie environment maintains program consistency, provides immediate type-checking feedback, and constrains the manipulation of the program to prevent syntax errors. JPie's integrated debugger, which uses the same graphical representation, allows logical errors (including exceptions) to be handled on the fly.

2 A SAMPLE APPLICATION

We illustrate JPie's features through the construction of a simple freehand drawing program in which a user can create strokes on a canvas with a mouse. Each stroke begins when the mouse button is pressed and ends when the button is released.

JPie encourages separation of the *data model* from the *view*. In our sample drawing application, the data model is the collection of shapes to be drawn, and the *view* contains a panel responsible for actually painting the shapes. To maintain the model, we define a *Draw* class that will be our "main" program. For use in the view, we also define a *ShapesPanel* class that extends Java's built-in JPanel class and overrides the *paint* method to draw a collection of shapes.

2.1 The *ShapesPanel* Class

2.1.1 Creating a Subclass

JPie's "Packages and Classes" window provides access to the classes and interfaces in the Java API, as well as to user-defined types. To create *ShapesPanel* as a subclass of JPanel, we select JPanel from Java's swing package, choose "Extend JPanel" from JPie's File menu, and specify the name and package of the new *ShapesPanel* class. At this point, JPie precompiles the *ShapesPanel* class so it can be treated by the JVM as an actual subclass of JPanel. In the *ShapesPanel* class window, the JPie programmer can manipulate the graphical representation to declare additional instance variables, as well as define and override methods and constructors.

2.1.2 Declaring Instance Variables

To hold the shapes to be painted, we declare an instance variable of type *java.util.Collection* by dragging that type from the Packages and Classes window and dropping it onto the "Data" panel. JPie automatically creates accessors and mutators (*get* and *set* methods, by the JavaBeans convention) whose names are kept consistent with the instance variable's name. Similarly, we declare a Boolean named *modified,* to be set true whenever the shapes collection is modified and therefore needs repainting.

2.2 The *Draw* Class

2.2.1 Constructors and Initialization

As the main program, we define a *Draw* class with a linked list of shapes as its data model. In the constructor, we call a reset method that instantiates a *java.awt.geom..GeneralPath* to hold the strokes of the user's drawing. We add the GeneralPath to the linked list after assigning it to a new instance variable.

2.2.2 Model/View Separation

In the "View" panel of the *Draw* class, we specify the visual appearance of its instances. Using drag-and-drop into the view, we create a *ShapesPanel* component for the drawing and a JButton below it labeled "clear." To link the model to the view, we establish a property connection from the list of the *Draw* class to the collection of the *ShapesPanel*.

2.2.3 Creating Instances

The "New instance" option in the "Instances" menu lets us manually instantiate an object of the *Draw* class. In the "Instances" panel, we can select and view any instance, whether created manually or programmatically.

2.3 Dynamic Changes

Having instantiated the *Draw* program, we proceed to modify it as it runs. We begin with painting and user event handling.

2.3.1 Overriding Methods

The *ShapesPanel* class window provides a summary list of all the methods declared and inherited. Each is represented as a capsule, showing its return type as an icon and its parameters as labeled slots. To paint the collection of shapes on the *ShapesPanel*, we override the *paint* method from JPanel by dragging the inherited method from the summary list and dropping it onto the methods panel. This dynamically modifiable method is then called polymorphically whenever the window system needs to repaint the component.

2.3.2 Editing a Method Body

Within the *paint* method, we dynamically define statements that first call the *paint* method inherited from the parent class and then iterate over the shapes collection to paint the shapes.

2.3.3 Adding a new instance variable and method

In the *Draw* class, we declare a new Boolean instance variable and define a *signalModified* method that sets the bit to signal the view when the drawing has changed. In the "View" panel, we establish a property connection from the new variable to the *ShapesPanel*'s modified property. Then, in the *ShapesPanel's setModified* method, we call *repaint* whenever true is assigned.

2.3.4 Event Handling

On the "Events" panel, we create event handlers to update the GeneralPath based on user input in the *ShapesPanel* component. We press the "record" button and demonstrate the desired event (*mousePressed* and *mouseDragged*). The system registers the listeners immediately. We edit the event handler method bodies to update the drawing in response to user actions. An event handler for the "clear" button is created similarly.

2.3.5 Adding a thread

The "Behaviors" panel lets us define threads that begin running immediately. To demonstrate this, we define a behavior that periodically changes the color of the drawing by calling *setForeground* with a new random color. JPie's thread -oriented debugger lets us observe and modify the execution.

3 CONCLUSION

JPie provides live interactive development of Java applications through direct manipulation of graphical representations of programming abstractions. All of this is accomplished without modification of the language or run-time system. Following nearly four years of development, we have begun using JPie to teach a "concepts first" introduction to computer science [2]. JPie is an ongoing project [1]. We invite participation from educators (for classroom use), as well as industry partners.

ACKNOWLEDGMENTS

We thank the following students for their contributions to the JPie project: Joel Brandt, Ben Brinkerhoff, Vanessa Clark, Melanie Cowan, Matt Hampton, Adam Mitz, Jonathan Nye, Sajeeva Pallemulle, and Richard Souvenir. We also thank the students in the spring semester of CS123 for valuable feedback. The JPie project is supported in part by a grant from the National Science Foundation.

REFERENCES

1. Kenneth J. Goldman et al., "JPie: Programming is Easy," http://jpie.cse.wustl.edu, July 2003.
2. Kenneth J. Goldman, "Washington University CS123: The Computer Science Way of Thinking," http://www.cse.wustl.edu/~kjg/cs123, January 2003.