Capsules and Semantic Regions for Code Visualization and Direct Manipulation of Live Programs

Kenneth J. Goldman
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130
(314) 935-7542
kjg@cse.wustl.edu

ABSTRACT
JPie is a visual programming environment supporting live construction of Java applications. Class modifications, such as declaring instance variables and overriding methods, take effect immediately on existing instances of the class to encourage experimentation in an educational setting. Because programs are edited live, editing gestures must transform the program from one well-formed state to another, without intermediate ambiguous states. To accomplish this, JPie's visual representation provides *capsules,* which represent logical code units, and *semantic regions,* which represent different aspects of a program. A capsule's meaning depends upon its containing semantic region. Similarly, a gesture, which involves manipulation of a capsule, is interpreted on the basis of the semantic region in which it occurs. This paper describes how capsules and semantic regions visually expose the structure of JPie programs and support live program editing through natural atomic gestures.

Categories and Subject Descriptors
D.1.7 [**Programming Techniques**] Visual Programming; D.2.6 [**Software Engineering]** Programming Environments – *graphical environments, interactive environments*

General Terms
Design, Human factors

Keywords
Live software development, program visualization

1. INTRODUCTION
The goal of our work is to make the power of general purpose software development accessible to a wider audience. To advance this goal, we have created JPie, an educational programming environment for live visual construction of Java applications.

In designing a programming environment for beginners, one option is to design a new language and execution model with a visual representation. However, one of our primary educational goals is to ease the transition from our visual environment into more traditional software development. We want students learn the semantics of a widely accepted programming language, and we want them to leverage that language in order to construct relatively sophisticated applications. Therefore, we did not invent a new programming language. Instead, we built JPie as a software construction *application* supporting the execution model of a standard underlying programming language (Java). JPie users have available the entire Java API, as well as third party Java classes. While providing access to a powerful general-purpose language, JPie is designed to provide a low entry barrier and a gradual learning curve.

JPie supports *dynamic classes* [14], which are like Java classes except that their class definitions can be changed at run-time. All changes to a dynamic class take effect immediately upon its existing instances. Examples of live modification include declaration of instance variables and methods, modification of method bodies, method overriding, user interface construction, and event handling. Dynamic classes embody executable data structures that represent the program. JPie users edit programs by manipulating a visualization of this executable data structure. The visual representation is designed to capture key abstractions of the programming model. Direct manipulation of the visual representation eliminates the need for textual source code, and live modification avoids the edit-compile-test cycle.

A common complaint of beginning programmers is forgetting the language syntax. At a superficial level, students may know that something is *possible*, but can't remember how to *write* it. At a deeper level, students often forget what is even *possible,* let alone how to write it. Therefore, one of the design goals for our visual representation was to expose not only what "is" (the structure of the existing program), but also what "could be" (the range of possibilities in a given context).

JPie's primary structural unit is the *capsule*. Capsules are simple graphical representations of programming language abstractions, such as variables and methods, that provide a unit of discourse for the programmer. To make software development immediate and tangible, capsules are used by direct

manipulation. Because each capsule represents an instance of some programming language concept, each gesture has a meaning in the programming model. This is different from textual programming, in which the unit of discourse is a character, and some editing operations result in nonsensical programs.

Capsules are manipulated within *semantic regions*. A semantic region is a labeled place that corresponds to some programming option within the language. For example, there are specific semantic regions for declaring methods, forming parameter expressions, and defining property connections. Semantic regions may be nested and may appear within capsules. For example, a variable declaration capsule contains semantic regions for its initial value and modifiers, whereas a constructor capsule would contain a semantic region for its super call and body statements.

Since they provide a scope or context, semantic regions roughly correspond to the empty space between delimiters in a textual programming language. However, they extend this idea in several ways. First, they are visually explicit so that programmers can see them as options, rather than having to remember the possibilities. For example, a box labeled "parameters" provides a clue to a beginning programmer, where a pair of parentheses might not. Second, semantic regions are more precisely constrained than lexical scopes. For example, there is one semantic region for declaring a method's local variables, and a different one for declaring the method's body statements. As we will see, this precision is advantageous for creating a simple vocabulary of editing gestures, since the location in which the gesture occurs can be used to discern the intentions of the programmer.

To support live experimentation, JPie lets programmers manipulate the program representation to effect changes while the program is running. Therefore, it is important that each gesture provide an atomic transformation from one well-formed program representation to another. Namely, there can be no intermediate ambiguous states in which the syntax is incorrect or in which declarations and uses do not correspond. Most atomic gestures, such as variable and method declaration and use, are accomplished by drag-and-drop. The capsule that is dropped and the semantic region into which it is dropped together determine the semantics of the gesture.

The remainder of the paper is organized as follows. Section 2 provides background on dynamic classes, which are the foundation for live execution in JPie and the back-end data structure for our visual representation. Section 3 presents JPie's mechanisms for visual representation and direct manipulation of dynamic classes using capsules and semantic regions. These are illustrated with a simple example in Section 4. Section 5 discusses JPie in relation to other work. We conclude, in Section 6, with a summary and directions for future work.

2. BACKGROUND

JPie rests upon on the concept of a *dynamic class* [14], whose members (variables, methods, and constructors) can be modified live, even while instances of that class exist in a running program. Internally, each dynamic class is modeled as mutable data structures that describe its members. For example, a dynamic method's data model includes a parameter list, local fields, and body statements. As we will explain, JPie provides a visual representation of the model, and direct manipulation of the visual representation results in corresponding changes to the data structures representing the members of the dynamic class.

Dynamic classes fully interoperate with compiled classes. Consequently, JPie programmers can use the entire Java API, 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 call their methods polymorphically.

To support interoperability, each dynamic class has a *compiled peer* that is generated when the dynamic class is created. The peer class allows dynamic classes (and their instances) to present themselves to the Java Virtual Machine (JVM) as any ordinary type. However, dynamic class 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. The implementation relies extensively on Java's reflection mechanism and is accomplished without modification of the language or the JVM.

The internal representation of dynamic classes directly captures the relationship between each use and its corresponding definition. For example, each variable read expression refers to the object representing the variable declaration, each method call refers to the method declaration, each actual parameter expression refers to the formal parameter declaration, each overriding method refers to the method it overrides, etc. Because the references are direct, traditional textual identifiers are not used to define the semantics of programs. Users can name variables and methods, but the names are used only for documentation purposes within JPie. When programs are exported for execution outside of JPie, the programming environment renames members when identifiers clash.

3. VISUAL REPRESENTATION AND DIRECT MANIPULATION

In any visual programming system, an important part of the design is the systematic creation of graphical representations for the programming abstractions. Our goal was to make as much information about each abstraction as explicit as possible, so as not to rely on the programmer's memory. Furthermore, we needed a representation amenable to direct manipulation. Finally, we wanted visual attributes the system could control for unobtrusive, timely feedback.

Screen space in visual programming systems is always at a premium. Wherever possible, we tried to provide ways to control the use of the space to help the programmer find and focus on the relevant portions of the program. Where space saving came into conflict with explicit representation, we gave explicit representation higher priority.

3.1. Access to Types

When JPie starts, the programmer sees a 'Packages and Classes' window, as shown in Figure 1. It contains a tree representation of all the packages and classes in the Java API (as well as the programmer's own classes) and a place to create shortcut panels for organizing frequently used classes into categories. From this window, one can open classes for editing and create new classes that extend other classes or implement interfaces.

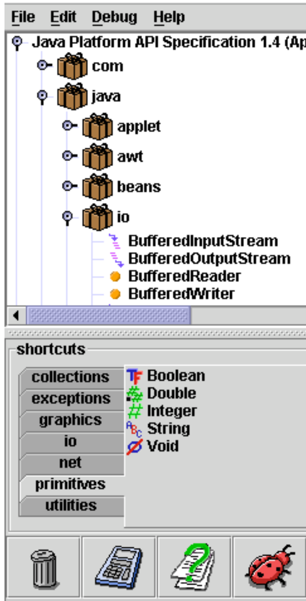

Figure 1. Packages and classes.

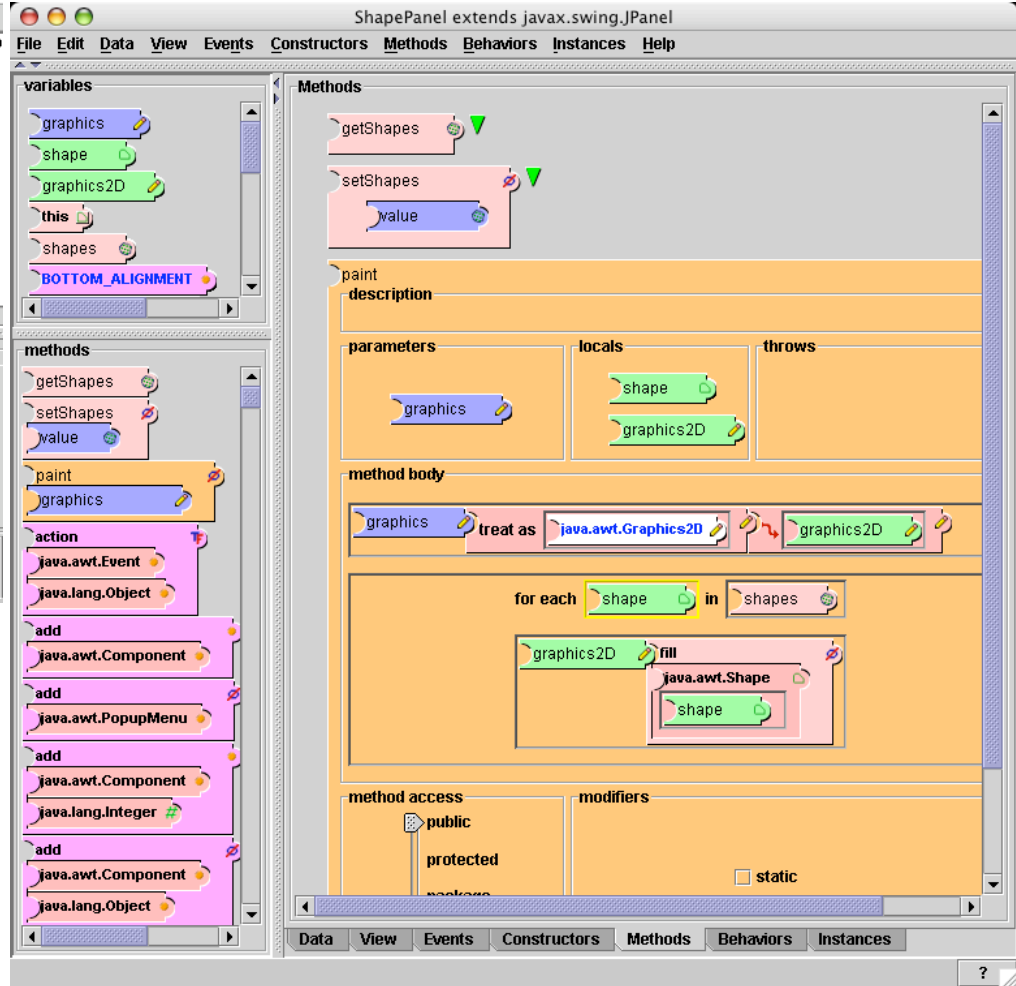

Figure 2. A class editing window, overriding "paint" in the methods panel.

3.2. Class Editing

Each class is edited in a separate window. Figure 2 shows a ShapesPanel class that extends the Java library class JPanel. Tabs along the bottom of the window provide access to panels for Data (instance variables), View (graphical appearance, layout, and property connections), Events (listeners to the view components), Constructors, Methods, Behaviors (periodic tasks that run as separate threads), and Instances (a list of instances of the class, selectable for viewing). Each dynamic class window also provides summary lists of the variables and methods of the class.

3.3. Capsules

JPie's principal visual unit is the *capsule*. Capsules represent types (classes), variable declarations, variable accesses, properties, methods, method calls, constructors, constructor calls, and can also encapsulate constants and expressions. Because they are represented visually, capsules afford an opportunity to present the program with more information than a textual identifier provides. For example, each capsule's type is shown iconically on a "dot" that protrudes from the right side of the capsule. The dot, which is reminiscent of dot notation in textual languages, provides an interaction point for making use of the capsule's value or result within expression. Every capsule also has a label, typically used to display a textual identifier. However, because declaration and use are linked by reference in the data structure, the identifier is inconsequential for execution within JPie. The color of each capsule indicates its scope. (For methods and instance variables, the color also indicates whether the member is declared or inherited.) Providing all of this information explicitly means that capsules occupy more space than a textual identifier, but it saves the programmer from having to rely on memory to know, for example, the parameter types and return type of a method call, or whether a variable is local or an instance variable.

Capsules are used by direct manipulation (selection or drag-and-drop), not indirectly by name. The system maintains consistency of identifier labels. If a variable or method is renamed, the labels at each use are updated.

3.4. Semantic Regions

Programmers manipulate capsules and other objects within clearly identified *semantic regions*. For example, all of the panels (Data, View, etc.) in the class editing window are semantic regions. Furthermore, a capsule can be opened by the

programmer to reveal semantic regions in which to edit the internal implementation of the abstraction represented by that capsule. For example, each method declaration capsule contains labeled regions for the formal parameters, local variables, method body, return expression (if the return type is not void), and access modifiers. Clicking within a method body creates a new statement at that point.

Semantic regions have two important advantages. First, they organize the program into structural units. This contributes to user interface consistency, efficient navigation and screen space utilization, and helps the programmer focus on the current task by controlling what information is displayed. Second, semantic regions serve as consistent and intuitive targets for direct manipulation operations, such as statement creation, expression building, and drag-and-drop operations.

For drag-and-drop gestures, the semantics and user feedback depend on what capsule is dragged, and into which semantic region it is dropped. Since a single drag-and-drop gesture conveys both an object and a target, relatively complex editing operations can be completed in one atomic gesture. For example, when a programmer drags a type capsule from the 'Packages and Classes' window into the Data panel, the system declares an instance variable of that type (and automatically defines associated 'get' and 'set' methods). On the other hand, if the programmer drags that same type into the Methods panel, the system declares a method with that return type. Similarly, dragging an inherited method into the Methods panel creates a method to override the inherited one. Additional examples for class and method capsules are shown in Table 1. Analogous gestures apply to other types of capsules, such as those respresenting variables, constructors, and the properties of graphical components. For example, similar gestures can be used to access a variable, reorder the formal parameter list, or move a local variable into the parameter list. All modifications, including method overriding, take immediate effect on existing instances.

3.5. Statements and Expressions

Statements and expressions are formed as chains of capsules. The bump on the right of a capsule, which indicates type of the expression at that point in the chain, provides opportunities to extend the chain (by accessing variables, calling methods, etc.), similar to the dot ('.') notation in textual programming. Nested boxes provide an explicit visual representation of scope and the order of execution. All execution, including assignment, occurs left to right, respecting the indicated nesting. This avoids any possible confusion about order of operations, and provides a convenient way to move statements among scopes. In addition, the capsules for method calls and constructor calls have semantic regions called *slots*, in which actual parameters are specified.

JPie provides a calculator-like interface for building statements involving control constructs (if, while, foreach, etc.) and expressions involving mathematical operators. Similar to [25], the constructs are provided as templates, but they contain specialized semantic regions. For example, dragging a class capsule into the loop variable region of a foreach statement results in the declaration of a local variable (of that type) used for iteration over the specified collection. See the paint method in Figure 2 for an example.

Table 1. Semantic regions determine the result of a drop.

A Class Capsule		**A Method Capsule**	
Dropped on	*Results in*	*Dropped on*	*Results in*
Shortcut panel	Shortcut	Method panel in same class	Move (reorder the method in the method list)
Data panel	Instance variable declared of that type	Method panel in an unrelated class	Copy of the method, and declaration of needed instance variables
Methods panel	Method with that return type	Method panel in child class	Overriding method
Parameter list	Parameter declaration	Statement	Method call
Locals panel	Local var.	Expression	Method call
Statement	Static reference to the class	Debugger	Breakpoint at that method
Foreach statement's variable slot	Local loop variable of that type	Trash	Deletion

3.6. Program Consistency

Maintaining program consistency during modification of a running program is a difficult problem. Because JPie's back-end consists of an executable data structure representing the program, its user interface can maintain program consistency while supporting a high degree of interactivity. The user interface prevents formation of syntactically incorrect statements and expressions by not allowing gestures that would result in them. For example, a variable can only be dropped into a scope having access to that variable. Similarly, when a chain is extended with a method call, only the accessible methods for that type are presented as options. User feedback is provided on a status bar, where each semantic region provides an explanation when it refuses a drop.

JPie also maintains referential consistency throughout the editing process. For example, when a variable's name is changed, all references to that variable (as well as the names of the 'get' and 'set' methods and their calls), are correspondingly updated. Similarly, when a parameter list is modified, the system automatically updates all of its method calls accordingly. Maintaining consistency is possible because the internal representation of dynamic classes directly captures the relationship between each *use* (variable reference, method call, actual parameter expression, etc.) and its corresponding *definition* (variable declaration, method declaration, formal parameter, etc.). An identifier is not represented as a text string, as it would be in a textual language. Instead, it is represented internally as an object that refers to the declaration of that variable.

JPie does not prevent two kinds of errors that may occur on the way to creating correct programs – missing expressions and type mismatches. For example, consider the semantic region (parameter slot) in which editing will take place for the actual parameter expression of a newly created method call. At first, the parameter slot will be empty. Then, since an expression of a

particular type is required, temporary type mismatches may necessarily occur along the way to forming the expression. If an integer is required as the parameter type and one drops a 'list' variable into the slot, there would be a type mismatch, but completing the expression by calling the 'size' method on the 'list' variable would fix the problem. Whenever an inconsistency occurs, the system provides immediate feedback. For example, each slot in a method call knows its expected type and will display a red border whenever its contained expression's type is not compatible. Incomplete expressions are similarly highlighted. References to deleted (or inaccessible) variables and methods are grayed out. In all cases, placing the cursor over the offending expression reveals pop-up text that explains the problem.

Because editing is live, an erroneous statement may be encountered during execution. However, rather than allow the program to fail, execution is suspended in JPie's debugger, where the programmer can correct the problem and resume execution.

3.2. Visualization for Debugging

JPie's thread-oriented debugger uses the same visual representation that is used in the class windows. Programmers can set breakpoints on methods, constructors, behaviors, event handlers, statements, and expressions. When a breakpoint or erroneous statement is reached within execution of a thread, a debugger window pops up, providing a visualization of the call stack as a series of tabbed panes. Each pane shows the expanded visual representation of the method (or other item) responsible for that stack frame. The debugger highlights (within each stack frame) the expression that is currently executing (or about to execute in the case of the top stack frame). In the debugger, the programmer can control the execution speed of that thread and watch the execution unfold, or can single-step through the execution expression by expression, with pop-up text displaying values for executed expressions. In addition to breakpoints, any consistency errors (such as type mismatches) cause the debugger to appear when execution of the offending expression is attempted. The programmer then has the opportunity to complete or correct the expression and resume execution.

Additional debugger support includes on-the-fly exception handling and detection of common logic errors. When an exception occurs that is not explicitly caught or thrown by a method, the debugger appears and provides the programmer with the opportunity to catch (or throw) the exception and resume execution. Proactive detection of logic errors includes dynamically adjustable stack bounding to detect possibly infinite recursion, dynamically adjustable loop bounding to detect possibly infinite loops, and deadlock detection. In the case of deadlock, a separate window appears with a visualization of the cycle in the wait-for graph. Within that visualization, the programmer can click on threads involved in the cycle in order to bring up debugging windows for them, and can optionally terminate them to break the deadlock. (See Figure 11.)

4. EXAMPLE

To illustrate JPie's support for visual representation and direct manipulation of live programs, we describe the construction of a simple timer application that displays a counter value on a button. The timer increments the counter value, once per second, and resets to 0 whenever the button is pressed. In the course of the example, we describe the steps one would take in JPie to create a dynamic class, create an instance of the class, declare and use an instance variable, define a graphical view and relate it to the data in the object, define and call a method, create a periodic behavior that executes as a separate thread, and define an event handler to react to a user event in the view.

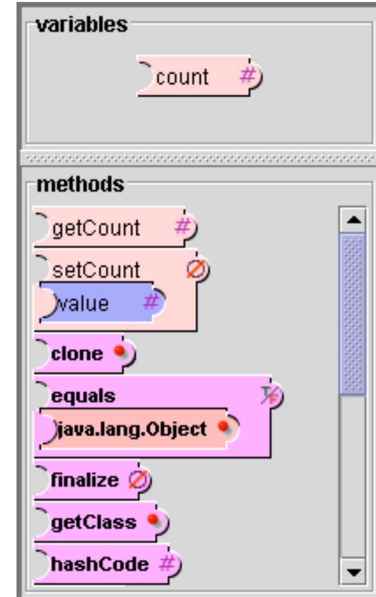

Figure 3. Instance variable and method summaries.

4.1. Classes and Instances

To create the timer application, we begin by creating a Timer class. Dynamic classes are created from the File menu, with the placement of the class in a folder corresponding to its Java package. One may specify that a class extends another class or implements an interface. Here, we simply allow the class to extend Object by default. Upon creation of the new class, JPie then generates the compiled peer class and opens the class window for editing.

JPie creates every dynamic class with a no-argument constructor whose body may be edited by the user. The constructor may be deleted, or parameters may be added, but as long as there is a no-argument constructor present, the class is instantiable from the Instances menu. All instances show in a list on the Instances panel, and can be selected there for viewing. Therefore, we may immediately create an instance of the Timer class. Until a view is defined for the class, there is nothing to see when an instance is selected. However, subsequent changes to the class definition, including the view definitions, will affect that instance in the running program, as will be seen in Figure 4 after we declare an instance variable and connect it to a graphical view.

4.2. Variables

Our first modification of the timer class is to create an instance variable that will keep track of the elapsed time. Because we want an integer variable, we drag the Integer class from the packages and classes window into the Data panel of the Timer class window. Dropping it there causes declaration of an instance variable, as well as creation of associated 'get' and 'set' methods. We rename the variable 'count' and the associated method names are updated accordingly, as shown in Figure 3. The methods summary list includes not only these declared methods, but also the inherited methods, in a different color. Each variable, method, and constructor is shown as a capsule, as described in Section 3. Recall that the variable type (or return type) is indicated iconically, and scope is indicated by color.

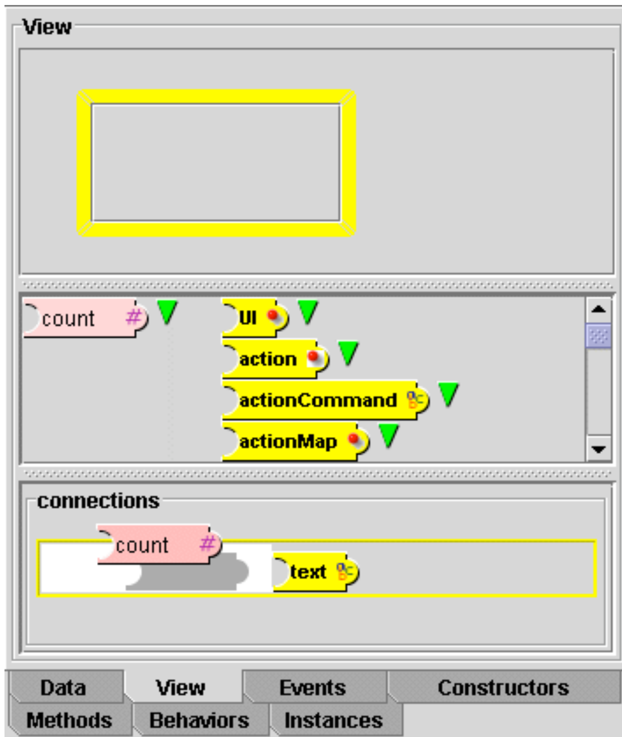

Figure 4. View construction.

4.3. Views and Property Connections

JPie provides a simple GUI builder, on the Views tab, that allows a view for a class to be created and associated with the data model. The functionality is not remarkably different from those provided by commercial tools. However, we mention it here for completeness, and also because its design is carried out as a logical extension of the capsule and semantic region metaphor.

Within the Views panel, one can drop capsules representing graphical component class types. This gesture results in instantiation of the component on the panel, where it can be subsequently manipulated for layout purposes. (JPie also provides some automatic layout options.) This arrangement of components specifies what each instance of the class should look like to the user. Selecting a component allows the programmer to see capsules representing the properties of that component. Those properties can then be connected to the properties of the class (i.e., instance variables) or to properties of other components in the view by dropping their capsules into a semantic region labeled "connections." In addition, initial values of component properties can be specified in semantic regions within expanded property capsules.

For our timer example, we drag the JButton class onto the Views panel, creating a button in the view. To make the value of the 'count' variable appear on the button in the view, we establish a property connection, again by drag and drop, that connects the 'count' property of each Timer object to the 'text' property of the JButton in each of its views, as shown in Figure 4. The change takes immediate affect on all instances, as shown in Figure 5.

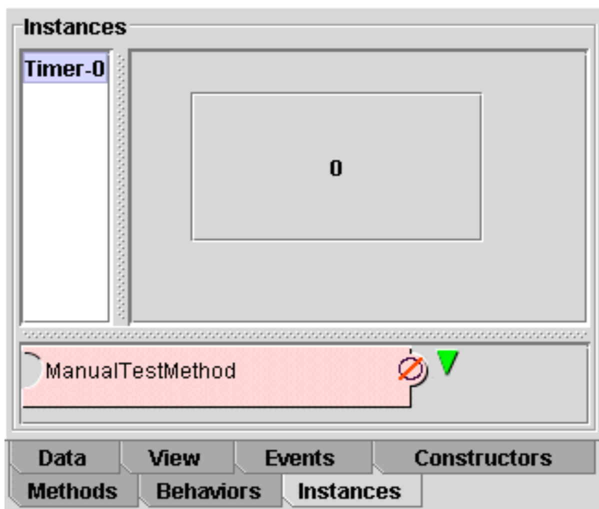

Figure 5. An instance of the Timer class.

4.4. Methods

We create a method for incrementing the count. As for variable declaration, we can declare a method by dragging the desired return type into the methods panel. Since the increment method will return nothing, we can either drag the Void type onto the methods panel, or choose 'New method' from the Methods menu to declare it. We name the method 'increment' and fill in its method body by dragging the 'setCount' method from the methods summary list into the method body. We then fill the parameter slot by dragging in the 'count' variable and adding 1 to it, as shown in Figure 6. We can test the method immediately in the Instances panel by calling it on an existing instance of the class, and see the text on the button change from 0 to 1.

4.5. Threads

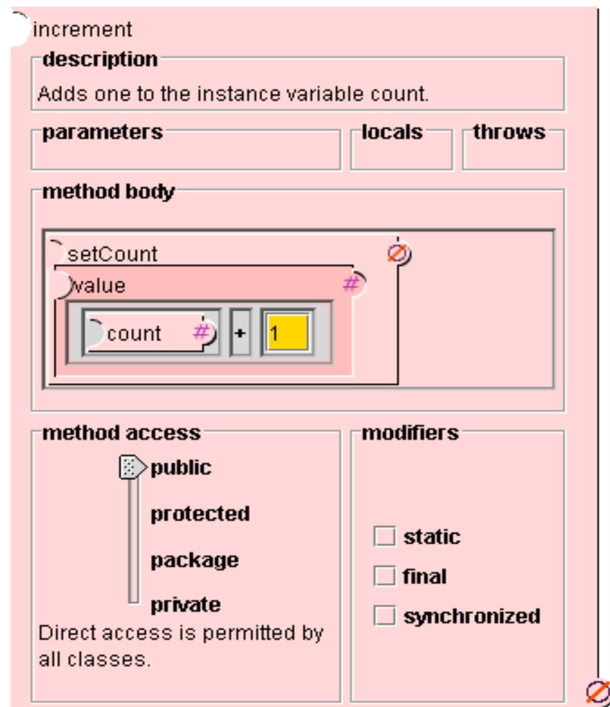

Figure 6. A method declaration.

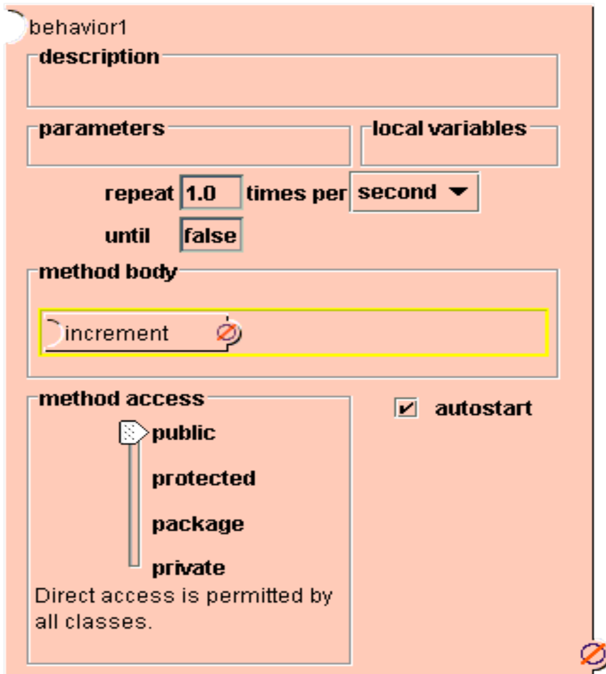

Figure 7. A behavior.

Programmers can create threads in the conventional way, by creating a subclass of the 'Thread' class and calling its start method. In addition, JPie provides a Behaviors panel to streamline the creation of threads that carry out periodic tasks within objects of the class. For example, instances of an 'animation' class might have a behavior that periodically changes the image in the animation. Each behavior looks like a method with a void return type, but has additional regions in which to specify a rate expression and a termination condition. Behaviors can be started automatically on instantiation.

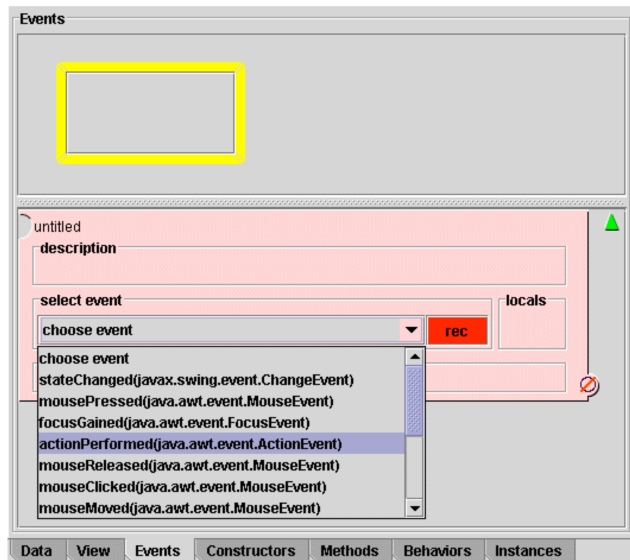

Figure 8. Recording and selecting an event.

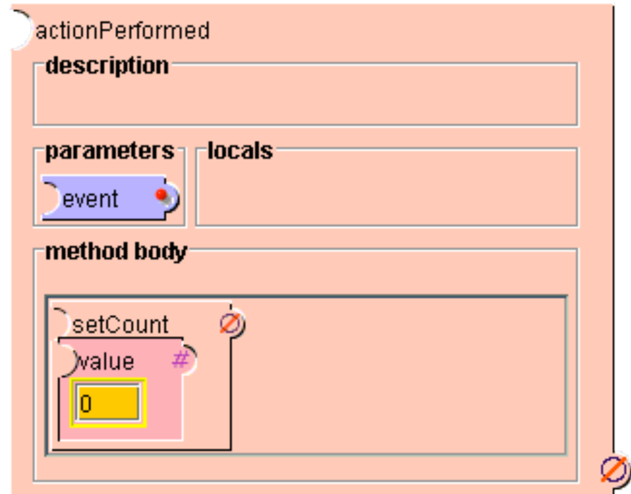

Figure 9. An event handler.

Since we want the Timer to increment once per second, we create a Behavior, which is a thread whose body runs periodically with a specified rate. In the behavior's body, which is edited just like a method body, we call 'increment' once per second, as shown in Figure 7.

4.6. Event Handlers

In the Events panel, components of a view can be selected to create event handlers that process user input. Each event handler is a listener method in the Java event model. The programmer demonstrates the user event of interest (mouse click, mouse entered, etc.) by performing the event on the selected component, and then selecting the event from a list of all the recorded events. At that point, the event handler method turns into the appropriate listener method with the appropriate parameter for that event type. The listener is automatically added to the component by the system so that whenever the event occurs in a view of any instance of the class, it triggers execution of the event handler within that instance. The programmer can edit the body of the event handler like any other method of the class.

To complete the timer example, we want a way for the user to reset the timer. To create an event handler, we select the JButton on the Events panel as the event source and choose 'New event handler' from the Events menu. We then press the record button in the event handler and begin acting upon the JButton. All events that we perform are recorded in a dropdown list, from which we choose the desired event, as shown in Figure 8. JPie immediately registers a listener for the event on the JButton in each view of each Timer instance. Finally, in Figure 9, we complete the example application by filling in the method body to reset the count whenever the user presses the button.

4.7. Debugging

Using the same visual representation as the class editor, the JPie debugger allows programmers to watch the execution unfold. Each thread is viewed in a separate window. The execution stack is shown as tabbed panes.

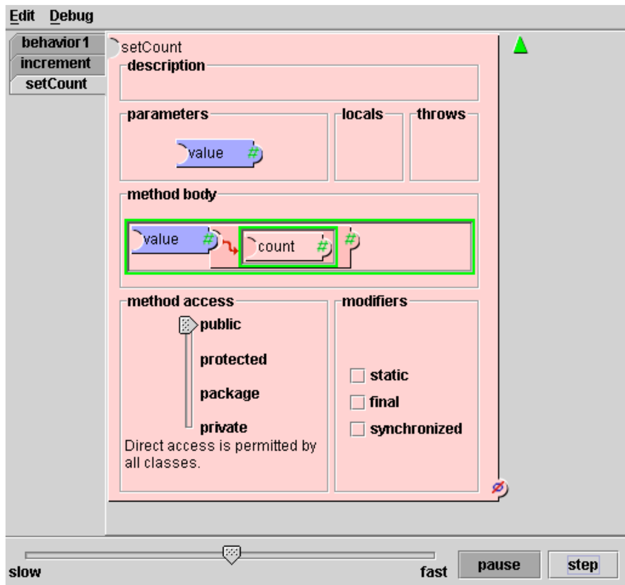

Figure 10. Watching execution in the debugger.

Figure 10 shows a debugging session for the thread that updates the counter. The tabs at the left show that the behavior called the increment method, which in turn called the setCount method. On each panel, the current step in the execution is highlighted in green. The user can pause and single-step the execution or watch it unfold in slow motion.

5. RELATED WORK

Prior work in making software development more accessible has generally taken three major approaches: high-level programming language abstractions, integrated development environments (IDEs), and visual programming systems.

5.1. High-level language abstractions

High-level languages like Fortran, C, Pascal, Smalltalk, Lisp, C++, and Java support abstractions that help make the programming process more natural. For example, languages provide constructs to express procedural abstraction, parameters and return values, abstract data types, encapsulation, iteration, objects, methods, class hierarchies, inheritance, and polymorphism. However, it is not the syntax itself, but the abstractions and programming model that account for the success of high-level languages. In fact, for the beginning programmer, learning how to encode these abstractions in the syntax of the language often becomes a distraction from understanding the abstractions themselves. In designing JPie, our goal was to leverage years of evolution in high-level language design by providing the abstractions and programming model of a modern language (Java). However, we wanted to eliminate the learning curve of textual syntax, offering instead direct manipulation of visual representations of programming abstractions within live programs.

5.2. Integrated Development Environments

Integrated development environments (IDEs) are the most common approach to facilitating programming in high-level languages. They are umbrella applications, combining project management, editor, compiler, run-time system, and debugger. Some support multiple programming languages. IDEs that support software development in Java include Inprise JBuilder, Sun ONE Studio, NetBeans, Eclipse, Webgain Visual Café, Metrowerks Code Warrior, and many others. Some sophisticated IDEs, such as Eclipse, allow functionality to be added to the IDE as third-party plug-ins [3]. Simpler IDEs specifically targeted for the classroom include BlueJ [2] and DrJava [1], which also provides some features as an Eclipse plug-in. IDEs support writing textual code in a number of ways. For example, source code editors may provide background compilation, syntax colorizing, delimiter matching, and method name completion. When compile or run-time errors occur, the environment highlights the line of text at which the error was detected. Common project management tasks are automated, and programmers do not have to separately invoke an editor, compiler, and debugger.

Educational IDEs, like BlueJ and DrJava, provide more interactive features than the typical IDE, such as manual invocation of methods on objects and interactive evaluation of Java expressions. JPie takes this interaction further by eliminating the need for a textual representation of the program. Furthermore, using a fine-grain internal representation of classes, JPie maintains global consistency throughout development, and allows the program to be modified while it is running.

Note that IDEs often do include a graphical user interface (GUI) builder for "live" layout of graphical components, where changes in relationships between the GUI and the underlying data model are reflected instantly. However, such live development applies only to certain aspects of the program. In the end, a programmer must write the essential functionality of the application in textual source code that is subsequently compiled and run. Moreover, although they are helpful for experienced programmers, GUI builders in IDEs can overwhelm inexperienced programmers because completing the application requires modifying computer-generated source code that beginning programmers are likely to find confusing.

5.3. Visual Languages

Our approach in designing JPie was to raise the level of abstraction by treating the programming process as an application domain, amenable to the same human-computer interface design principles that have been applied successfully to other types of applications. Central to these principles is direct manipulation of domain-specific entities. This is not a new idea. Research in visual languages has produced many systems supporting direct manipulation of program entities, but the emphasis of that research has been on finding new ways to think about computation that are particularly well-suited for visual expression. Visual programming environments are typically based on *new* languages (as opposed to being graphical front-ends for existing languages), and they are generally based on execution models that are deemed to be particularly well suited for visual expression. These often provide a tight integration of editing and program execution.

One of the most common visual language paradigms is dataflow, in which data flows across "arrows" to trigger actions performed in "boxes." Examples include Show and Tell [8], one

of the first dataflow languages designed to be accessible to children; Prograph [7], which has been developed commercially; Khoros [22], which has been targeted for image and signal processing; and our own distributed application configuration language [19]. However, dataflow is not the only visual language paradigm. Forms/3 [5] introduces procedural abstraction within a declarative programming spreadsheet paradigm. ThingLab [4] uses a constraint-oriented paradigm to support the construction of geometric models. Statecharts [16] supports software development with a nested state-machine paradigm. VIPR [6] uses arrows and nested rings to declare program behavior with elements of object-oriented programming. AgentSheets [23][24] provides a rule-based paradigm for specifying how interacting agents gather and process information. Stagecast (a commercial realization of KidSim [9] and Cocoa [17]) uses a combination of rule-based programming and programming-by-example to support children in developing games and simulations on a graphics grid. Logo [20], a well-known programming language for children, uses a mixture of visual components and textual programming.

Among visual languages, Alice [21] has a visual representation that is closest to JPie's. Alice appeals to children, who can use direct manipulation of program abstractions to construct object-oriented applications based on 3D worlds. Alice is not designed to support general-purpose programming, but instead streamlines programming for the 3D graphics domain. In Alice, objects are instantiated before the program runs in a global object tree. Therefore, the "main" program, methods, and event handlers refer to particular named instances.

Although it does not support editing of live programs, Alice's visual representation and user interface strictly enforces program consistency. For example, when creating a method call, Alice requires programmers to fill in required parameters as part of the gesture. However, in doing so it places some limitations on expressive power. For example, one cannot temporarily use a variable of the wrong type as an actual parameter, even if one intends to subsequently call a method on that variable to form an expression of the correct type. Like JPie, Alice uses templates for programming constructs. Drag-and-drop is used with copy/paste semantics. Alice provides semantic regions to the extent that a variable can be dropped into an expression to create a reference to that variable, and a method can be dropped into a statement to create a method call. Alice does not provide access to an underlying general-purpose programming language.

In summary, JPie differs from prior work in visual languages by simultaneously achieving several goals. First, rather than create a new language, we chose to provide a visual representation for writing sophisticated applications in a widely accepted programming language. Second, we created an open system that allows access to the full Java API and other compiled classes, with interoperability and full polymorphism. Finally, we provide capsules and semantic regions to support live modification of running programs through atomic gestures.

6. CONCLUSION

JPie provides graphical representations of programming language abstractions that expose the Java execution model and make software development immediate and tangible. JPie programmers directly 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. Through dynamic classes, JPie has fine-grain awareness of program structure and takes advantage of this knowledge to constrain program editing, check and maintain consistency, provide timely feedback, and eliminate the edit-compile-execute cycle. JPie's debugger uses the same graphical representation to allow logical errors to be handled on the fly.

Under development since 1999, JPie has been used for three semesters in Washington University CS123, a laboratory course that introduces object-oriented software concepts to undergraduate students without prior programming background [10][15]. Using JPie's visual representation, students are introduced to object-oriented programming, and because the course does not have the usual programming language learning curve, by the end of the semester, the students are using JPie to construct relatively sophisticated Java applications. For example, in one assignment, students create an internet chat application. Using only standard Java sockets, the students build both the client and a multithreaded server.

We are currently working on additional features, such as heap visualization and learning curve management, to enhance JPie's pedagogical value. Educators interested in using JPie in the classroom are encouraged to contact the author or see the JPie web site [13].

7. ACKNOWLEDGMENTS

I thank the following students for their contributions to the JPie project: James Aguilar, Ben Birnbaum, Joel Brandt, Ben Brinckerhoff, Vanessa Clark, Melanie Cowan, Asha Haji, Matt Hampton, Dylan Lingelbach, Adam Mitz, Brandon Morgan, Jonathan Nye, Sajeeva Pallemulle, Joyce Ann Santos, Richard Souvenir, Ray Thomas, and Haraldur Thorvaldsson. I also thank all the students who have taken CS123 for valuable feedback. This work was supported in part by the National Science Foundation under CISE Educational Innovation grant 0305954.

8. REFERENCES

[1] Eric Allen, Robert Cartwright, and Brian Stoler, "DrJava: A lightweight pedagogic environment for Java," *33rd SIGCSE Technical Symposium on Computer Science Education*, February 2002.

[2] David J. Barnes and Michael Kölling, *Objects First with Java: A Practical Introduction Using BlueJ.* Prentice Hall/Pearson Education (2003).

[3] Erich Gamma and Kent Beck, *Contributing to Eclipse,* Addison Wesley, Boston (2004).

[4] A. Borning, The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, ACM Transactions on Programming Languages and Systems (1981), vol. 3 pp.355-387.

[5] Margaret Burnett, John Atwood, Rebecca Walpole Djang, Herkimer Gottfried, James Reichwein, and Sherry Yang "Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm," *Journal of Functional Programming*, 11(2), 155-206, March 2001.

[6] W. Citrin, M. Doherty, and B. Zorn, Formal Semantics of Control in a Completely Visual Programming Language, Proc. of 1994 IEEE Symposium on Visual Languages, St. Louis, (1994), 208-215.

[7] P.T. Cox, F.R. Giles, and T. Pietrzykowski, (1989). ``Prograph: a step towards liberating programming from textual conditioning." 1989 IEEE Workshop on Visual languages, pp. 150-156.

[8] T.D. Kimura, J.W. Choi, and J.M. Mack, “A visual language for keyboardless programming.” Technical Report WUCS-86-6, Washington University in St. Louis, June 1986.
[9] Allen Cypher and David C. Smith, "KidSim: End User Programming of Simulations." In Proceedings of CHI, 1995 (Denver, May 7-11). ACM, New York, 1995, pp. 27-34.
[10] Kenneth J. Goldman, “A Concepts-First Curriculum for Introductory Computer Science,” *35th SIGCSE Technical Symposium on Computer Science Education*, March 2004.
[11] Kenneth J. Goldman, “A Demonstration of JPie: An Environment for Live Software Construction in Java," *OOPSLA'03 Conference Companion*, October 26-30, 2003, Anaheim, California, USA.
[12] Kenneth J. Goldman, “An Interactive Environment for Beginning Java Programmers,” *Science of Computer Programming,* Special Issue on Practice and Experience with Java in Education, Elsevier, 53 (2004) pp. 3-24.
[13] Kenneth J. Goldman et al., “JPie: Programming is Easy,” http://jpie.cse.wustl.edu, July 2003.
[14] Kenneth J. Goldman, “Live Software Development with Dynamic Classes,” *submitted for publication.*
[15] Kenneth J. Goldman, “Washington University CS123: Introduction to Software Concepts,” http://www.cse.wustl.edu/~kjg/cs123, January 2003.
[16] David Harel, “Statecharts: A Visual Formalism for Complex Systems.” Science of Computer Programming, 8(3):231-274, June 1987.
[17] N. Heger, A. Cypher, and D.C. Smith, "Cocoa at the Visual Programming Challenge '97." In Journal of Visual Languages and Computing 9(2), 1998. pp. 151-169.
[18] Alan Kay, “The early history of Smalltalk.” In T.J. Bergin and R.G. Gibson, eds., *History of Programming Languages - II*, New York: ACM Press and Addison-Wesley Publishing Co., 1996, pp. 511-578
[19] T. Paul McCartney and Kenneth J. Goldman. “Visual Specification of Interprocess and Intraprocess Communication.” In Proceedings of the 10th International Symposium on Visual Languages (VL'94), St. Louis, MO, October 1994, pp. 80-87.
[20] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books: New York, 1980.
[21] Randy Pausch, Tommy Burnette, A.C. Capeheart, Matthew Conway, Dennis Cosgrove, Rob DeLine, Jim Durbin, Rich Gossweiler, Shuichi Koga, Jeff White, *Alice: Rapid Prototyping System for Virtual Reality*, IEEE Computer Graphics and Applications, May 1995.
[22] J. Rasure and C. S. Williams, "An Integrated Visual Language and Software Development Environment," Journal of Visual Languages and Computing, Vol. 2, 1991, pp 217-246.
[23] A. Repenning, "Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments," University of Colorado at Boulder, Ph.D. dissertation, Dept. of Department of Computer Science, 1993.
[24] A. Repenning, "Creating User Interfaces with Agentsheets," 1991 Symposium on Applied Computing, Kansas City, MO, IEEE Computer Society Press, 1991, pp. 190-196.
[25] Tim Teitelbaum , Thomas Reps, The Cornell Program Synthesizer: a syntax-directed programming environment. Communications of the ACM, 24, 9 (Sep. 1981), 563-573.

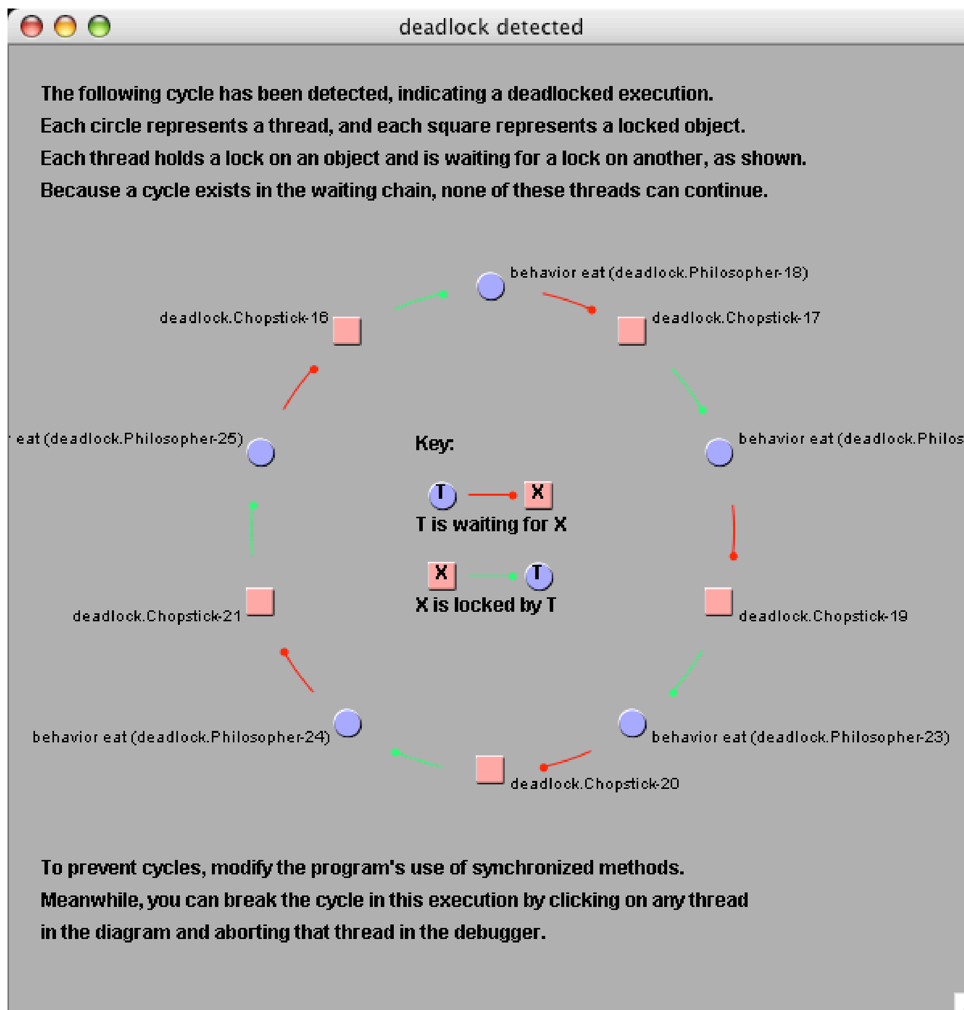

Figure 11. Deadlock visualization shows the threads and objects involved in a cycle.