Methodology and Architecture of JIVE

Paul V. Gestwicki, Bharat Jayaraman
[pvg|bharat]@cse.buffalo.edu
Department of Computer Science and Engineering
University at Buffalo, State University of New York
201 Bell Hall, Box 60200
Buffalo, NY 14260-2000

December 13, 2004

Abstract

A novel approach to the runtime visualization and analysis of object-oriented programs is presented and illustrated through a prototype system called JIVE: *J*ava *I*nteractive *V*isualization *E*nvironment. The main contributions of JIVE are in its multiple concurrent representations of program state and execution history through an environment that supports forward and reverse execution along with execution queries. This model facilitates program understanding and interactive debugging. Our graphical representation of runtime states clarifies the important point that objects are execution environments. The history of object interaction is displayed via sequence diagrams similar to those of the Unified Modeling Language, and in this way we help close the loop between design-time and run-time representations. Interactive execution is made possible by maintaining a runtime history database, which may be queried for information on variable behavior, method executions, and object interactions. We illustrate the capabilities of this system through examples. JIVE is implemented using the Java Platform Debugger Architecture and supports the Java language and libraries, including multithreaded and GUI applications.

1 Introduction

This paper presents several novel techniques for enhancing runtime comprehension of object-oriented programs. Object-oriented programs differ from procedural programs in two important ways: (i) objects are not just data structures, but serve as environments within which procedure activations take place; (ii) object-oriented programs engender the use of smaller methods, and more complicated interactions among objects result. Runtime comprehension is therefore enhanced by providing views of the object structure as well as the history of object interaction. Although these observations are fundamental, to our knowledge there is no visualization system for object oriented programs that realizes these basic criteria. As we will explain, it necessary for the success of such a system to have multiple views of the object structure and to couple these with views of execution history.

Just as graphical notations such as UML [3] clarify high-level relationships at design time, a graphical depiction of the relationships among objects and method activations at runtime is highly desirable[8]. The visualizations presented in this paper are applicable to object-oriented languages in general, but we focus on Java in particular to show that our methodology can handle the nuances and complications of a versatile language used for diverse applications.

We identify the following major requirements for a system that visualizes runtime states of Java:

1. *Depict Objects as Environments.* As noted earlier, the execution states of object-oriented programs differs fundamentally from those of procedural programs since an object is an environment within which method activations take place. This is exemplified in the tools surveyed as related work in Section 2 that clearly depict method call sequence and that support inspection of objects' internal details. However, these tools depict neither the overall object structure nor the method activations within these objects, and hence important relationships are missing in

the visualization.

2. *Provide Multiple Views of Execution States.* The current execution state of the program should be observable at varying levels of granularity for better comprehension. A novice user may wish to see abstract, simplified relationships between runtime components, whereas an expert might look for the specific details of an inheritance hierarchy or complex data structure. This allows the visualizations to be useful for both teaching and debugging, and it facilitates use by users with various levels of experience.

3. *Capture History of Execution and Method Interaction.* The history of program execution should be observable using notations such as time-sequence or collaboration diagrams[3]. While these diagrams were motivated by program design considerations (to document the details of use cases), the ability to produce such diagrams at runtime helps close the loop between program design and program execution. Additionally, the visualization of program history should be interactive, allowing the user to select the point in program history that he or she wishes to view. For example, selecting a method activation in a sequence diagram should cause the visualization tool to show the execution state at which the method was called.

4. *Support Forward and Backward Execution.* It should be possible to interactively step forward or backward through program execution. This capability is especially important in debugging, since the occurrence of an error is usually detected after the point of error[1]. The user should also be allowed to decide the granularity of stepping, for example, through statement-level and method-level step sizes or by setting execution breakpoints. Moreover, these capabilities should also be supported for multithreaded programs.

5. *Support Queries on the Runtime State.* One of the most important requirements for program debugging is understanding how the variable values are changed. It should be possible to query the runtime state for properties of variables, such as when a variable changed or took a certain value. This requirement in turn forces one to view the runtime state as a database that can be queried.

6. *Produce Clear and Legible Drawings.* The visualization environment should automatically arrange diagram components so as to clarify the object structure and method-calling sequence. Custom visualizations of commonly used types such as arrays, lists, and tables should be provided. Patterns inherent in the runtime structure, such as shapes of known data structures, should also represented in an intuitive manner.

7. *Support Large Object Structures.* The system should be capable of visualizing small and large applications with comparable effectiveness and with reasonable resource requirements. The scalability property applies to both the system resources required for visualization as well as the interface to the visualization environment. Visualizations should be navigable and customizable so that user productivity is optimized.

8. *Use Existing Java Virtual Machine.* It is important for the visualization system to run on on existing Java Virtual Machines (JVM) and not require a custom implementation of a Java interpreter. A custom JVM implementation will be hard-pressed to keep up with advances in Java technology; for example, the new syntax supported in Java 2 Standard Edition 1.5 would require changes to any custom compilers and interpreters[29]. Additionally, it should be possible to visualize programs with graphical user-interfaces built from libraries such as Swing and AWT.

We have used these criteria to guide our research in effective means of visualizing execution states and runtime details. The fundamental steps required to develop a visualization environment in keeping with the above requirements are: creating a visual operational semantics for Java; developing a model for interaction and reverse execution; generating multiple versatile and customizable views of runtime state; and integrating these into an application framework. The resulting tool would be usable as a visual debugger and as a teaching aid. To these ends, we have created a prototypical tool called *JIVE: Java Interactive Visualization Environment.*

The remainder of this paper is structured as follows: Section 2 surveys the related work in order to give the context and motivation for JIVE; Section 3 describes in general how and why we provide multiple, interactive, customizable views of Java program states and execution history; Section 4 describes the specific changes made to our visualization notation in order to support the nuances of Java runtime semantics; Section 5 gives an overview of the software architecture of JIVE, specifically focusing on its modular design and capacity for interactive execution; Section 5 also describes our view of runtime history as a database that can be queried for information; and Section 6 summarizes our contributions and outlines areas of further research.

2 Related Work

In the previous section, we outlined the requirements for the interactive visualization of Java. To the best of our knowledge, there is no tool that meets all of these criteria, though there are many projects that address one or more of our requirements. JIVE integrates a broad range of ideas, and so we organize the related work into categories below. A few prominent projects from each category are identified, and their relationship to this work is discussed.

Integrated Development Environments. JIVE is not a programming environment, but its debugging capabilities merit comparing it to the debuggers present in common IDEs. Common development environments such as Eclipse and Microsoft VisualStudio provide integrated debugging tools, but despite advances in development tools, these debuggers are still intrinsically based upon the classic `dbx`. The interfaces are based on interactions of text rather than interactions among graphical components, and so important relationships among methods and objects are lost. Our preliminary user studies have revealed that our notation is significantly more useful to students who are learning the object-oriented methodology. The simple list and tree visualizations of DDD[33] (a front-end to `dbx`) also do not capture the concept of objects as environments. The Smalltalk Inspector[14] provides a view of objects similar to the visual semantics we propose. However, it does not provide a view of program history, nor does it visualize object graphs or methods in their inherited object contexts.

Visualization Tools for OOP. There is a branch of related work that is dedicated to pedagogic applications of program visualization. One such tool is jGRASP[17], which is a development environment designed for educational use. It has integrated a true visual debugger, and it provides dynamic state visualization rather than animation of specific algorithms [4]. jGRASP provides intuitive graphical representations of program design and runtime structure, but it does not provide a visual operational semantics for Java; the visualizations are intentionally abstract in order to be understood at a high level. Also, jGRASP does not include support for interactive execution in the reverse direction, and there cannot provide facilities for comparative analysis of program histories or querying facilities.

Another prominent visual tool for teaching Java programming is *BlueJ*[20]. The main visualization feature of BlueJ is its ability to program through the creation of interactive class diagrams. BlueJ solves many common problems of teaching the objects-first approach to computer science, but BlueJ does not highlight the important fact that objects are environments of program execution: it does not display structural relationship between objects, and it does not provide a visualization of method activations in their appropriate object environments. BlueJ helps teach the objects-first approach, but it ignores some low-level details of scoping and the complex interactions of objects at runtime. While some notable researchers have discouraged visualizations at JIVE's level of detail[9], we believe that our method of dynamic, interactive drawings provides the most complete basis for a deep understanding of runtime states, and our model of multiple visualizations at varying levels of detail addresses the problem of information overload.

Other related work is focused more on visualization than on pedagogic applications. The cel visualizations of Walker *et al.* show a high-level model of object-oriented execution [31]. This technique requires the engineer to specify what information is visualized, and it is designed for the dynamic environment of program execution. Our approach shows a much lower-level visualization, clarifying individual states and object interaction in history, rather than high-level behavior such as class-level abstractions [25]; our intent is significantly different despite outward similarities.

Interactive Execution. Reversible execution has been explored in two general forms: re-execution[32] and state-saving[2]. The re-execution model uses repeated executions of a program to reach the desired point in execution. This model has the advantage that there is not much data to be logged, but it has the obvious drawback that execution must be repeated each time a single backward step is made. JIVE uses the state-saving model, which has the drawback that transaction logs grow quickly. However, increased processor speed and decreased cost of memory continue to reduce the overhead of our models.

JIVE's state-saving mechanism uses a form of declarative event analysis for object-oriented programs [26], though the events themselves are tailored for integration with JPDA [28]. Our model of recording program execution allows for queries that are not just on specific states [24], but also over execution history. However, this technique precludes the possibility of effective memory-usage and time-efficiency analysis, since JIVE imparts certain operational costs on execution, described in more detail in Section 5.

Our Previous Work. This work described in this paper considerably extends our previous work in interactive visual execution of Java. Our earlier work[12] served as a proof of concept that a modified contour model of execution can be used to visualize Java execution, and furthermore that such a model supports interactive execution (forward and reverse stepping through execution states). This earlier model was based on a source-code transformation that converts an arbitrary Java program into a self-visualizing Java program. We have since altered our model to one that does not use source-code transformation; this advancement frees us from the task of writing complicated parsers that are beholden to the evolving Java grammar[29]. Instead of transforming the source code, we use the Java Platform Debugger Architecture (JPDA)[28] to initiate communication between JIVE and the program being visualized and to monitor the program for events that affect the visualization.

Our current visual semantics for Java is enhanced from our previous work; we have extended the notation to support static contexts, inner classes, threads, and Java's unique overriding and shadowing behavior. We have further enhanced our model by providing multiple views of the execution state at customizable levels of detail. The history of program execution is shown using interactive sequence diagrams that allow the user to inspect arbitrary execution states without having to step forward or backward to them. Another considerable advance is the view of program execution history as an object-oriented database: it is possible to query the execution history for information on methods, variables, and structures, and multiple execution histories can be compared. This database view of program execution is made possible by recording key events during execution [26].

3 Multiple Views of Execution

In our earlier work, we showed that an extension of the *contour model* [19] provides a clear depiction of the execution of object-oriented programs[12, 18]. Contour diagrams were introduced for understanding recursion and scope rules in statically-scoped procedural programming languages such as Algol and Pascal. In essence, the classical contour diagram consists of a set of nested rectangles, or *contours*, where each one represents the runtime information of the activation of a procedure-level construct. This runtime information can include the bindings for parameters and local variables, the source code associated with the construct, the executable instructions, and appropriate linkage information. Traditional contour diagrams are insufficient to capture the history of execution, and large diagrams are too detailed to facilitate the understanding of runtime states. In order to address the first problem, we provide multiple, customizable, abstracted views of the program state that emphasize different aspects of execution. To depict execution history, we employ UML time sequence diagrams[3].

We will consider some sample visualizations before exploring details of contour semantics for Java in Section 4. A partial implementation for a binary search tree is given in Figure 3. `BST` is a simple binary search tree, and `DupTree` is a subclass that counts duplicate entries. This example may be deceivingly simple to a reader who is familiar with object-oriented methodology, but this basic program highlights our focus on objects as environments. The `insert` method of `DupTree`, for example, refers to a member variable `data` through a reference to `this`, but there is no `data` member defined in the `DupTree` class. Understanding how `this` and `data` are resolved requires an understanding of inheritance and variable shadowing. Our visualization methodology is able to clarify the relationships among these features of object-oriented languages in general and Java in particular.

Compact Views

The given binary search tree is structurally recursive: each branch of the tree is itself a subtree. Figure 2 gives one possible visualization of a state from the program's execution. The top portion of the screenshot shows a JIVE contour diagram visualization of the object structure. We call this a *compact view* since not all of the details of each contour are shown. The nodes of the tree are shown as dark-bordered

```
public class BST {
  private int data;
  private BST left, right;
  public BST(int data) {
    this.data=data;
  }
  public void insert(int data) {
    if (data < this.data)
      if (left==null)
        left = createNode(data);
      else
        left.insert(data);
    else if (data > this.data)
      if (right==null)
        right = createNode(data);
      else
        right.insert(data);
  }
  protected BST createNode(int data) {
    return new BST(data);
  }
}

public class DupTree extends BST {
  private int count = 1;
  public DupTree(int data) {
    super(data);
  }
  public void insert(int data) {
    if (this.data == data) count++;
    else super.insert(data);
  }
  protected BST createNode(int data) {
    return new DupTree(data);
  }
  public static void main(String[] args) {
    DupTree t = new DupTree(10);
    t.insert(5);
    t.insert(7);
    t.insert(3);
    t.insert(15);
    t.insert(18);
    t.insert(5);
  }
}
```

Figure 1: Partial implementations of `BST` and `DupTree` in Java. `BST` is a simple binary search tree for integer data, and `DupTree` is an subclass of `BST` that counts duplicates.

instance contours. Each is named for the class of which it is an instance (in this case, `DupTree`), and an *instance count* is added to distinguish individual objects. For example, `DupTree:1` is the first instance of the `DupTree` class. The instance contours make up the *instance space*; the *static space* is made up the light-bordered *static contours*. There is one static contour for each class loaded by the Java system. A static contour is nested within its superclass' static contour, and so since all classes in Java are subclasses of `java.lang.Object`, all static contours are nested within the static contour for `java.lang.Object`. The complications and implications of the instance/static dichotomy will be explored in Section 4 along with other Java-specific concerns. The arrows in the diagram are *structural links*, which indicate that there is a reference from one object to another. These links show the connection between nodes that form the binary tree structure, and they correspond directly to the `left` and `right` member variables of `BST`.

Sequence Diagrams

The lower portion of Figure 2 shows the history of execution through a sequence diagram; the visible portion shows the beginning of the this program's execution. Both the static and object contours are shown as contexts along the top of the sequence diagram since both are environments of method execution. The method activations themselves are shown as rectangles along the vertical lifelines of their respective contexts. The `main` method is the leftmost method activation, and it is responsible for the creation of the first three tree nodes, `DupTree:1`, `DupTree:2`, and `DupTree:3`. The methods labeled "`<init>`" indicate the creation of the new context by calling its constructor. Java uses this symbol internally to refer to constructor and instance initializer invocations, and so we adopt the notation here. Since JIVE is an interactive environment, we allow the user to scroll through the sequence diagram and select contexts or method activations, and JIVE will jump to the contour diagram visualization for those states.

Detailed Views

A *detailed view* is shown in Figure 13 (at the end of this document). The detailed view exhibits how inheritance is shown in our visualizations: each instance of `DupTree` is wrapped in an instance of `BST`, and each `BST` is in turn wrapped in an `Object`. Containment hierarchy visualization of inheritance is

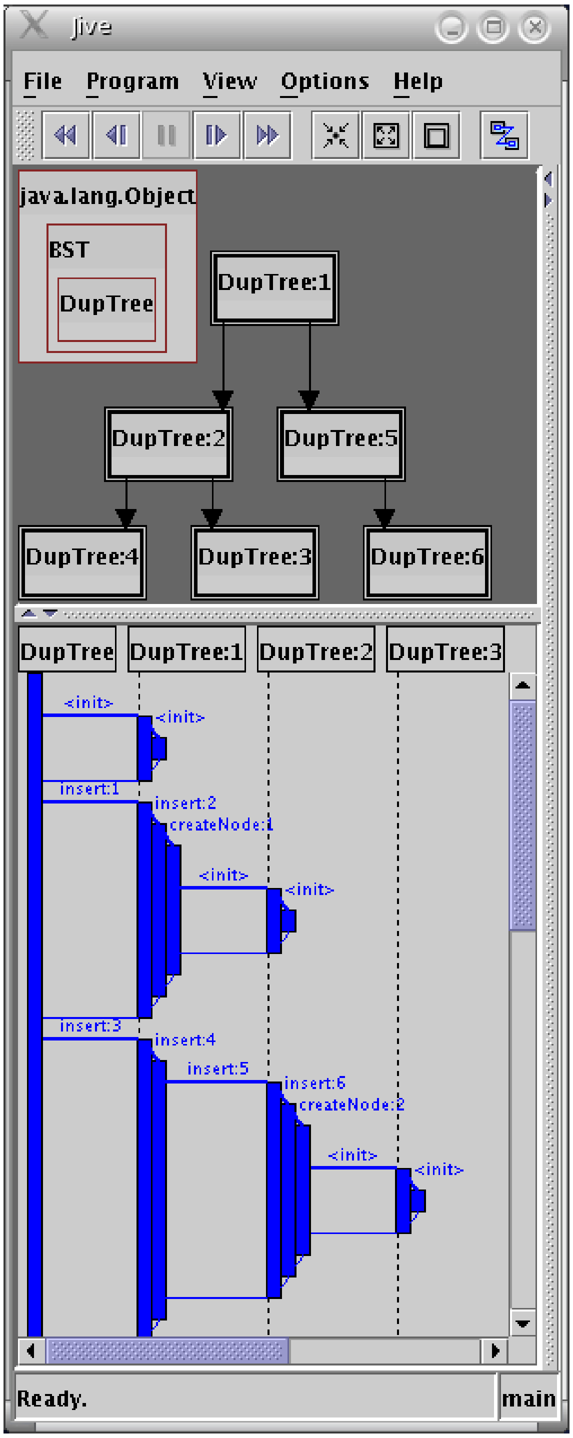

Figure 2: Compact view of a binary search tree with partial sequence diagram. In the compact view, objects are labeled by their most specific class, and method activations are not shown.

only applicable for object-oriented programming languages that are limited to single-inheritance. Java's single-inheritance of classes fulfills this requirement. Multiple inheritance of interfaces in Java does not invalidate this methodology since interfaces do not have contour representations.

The *member tables* of each instance contour have also been expanded in this diagram. These tables show the variables defined within the contour's context; the tables can be configured to show method definitions as well, but we have found this to take inordinate space in the visualization. It is not generally necessary to show the member tables of every contour, but it is done in this figure in order to demonstrate the highest level of detail. The screenshot also shows specifically how structural links are built between contours: the link starts at the value cell of the member table and is drawn to the contour being referenced. This construction allows structural links to properly implement static variable scoping with inheritance. For example, in an `insert` method activation `DupTree`, the expression "`node.data`" is resolved as follows: follow the link from `node` in the member table to the appropriate `DupTree` contour; look for `data` in the `DupTree` contour; and then look for `data` in the `BST` contour surrounding the `DupTree` contour, where an appropriate entry is found. This search strategy for symbols is an application of traditional contour model semantics[19] and is not specific to object-oriented programs, but it does illustrate how naturally the model is applied to object-oriented environments [18].

Method contours represent method activations, and a visualization with method contours can be found in Figure 14 (at the end of this document). Method contours have a member table that contains the local variables of the method, including parameters, along with the return point and dynamic link, abbreviated `rpdl`. The member tables of method contours, like those of instance and static contours, may be hidden or shown on demand. Method contours are nested within their defining contexts. The figure shows how the `insert` method in `DupTree` calls the `insert` method in its surrounding `BST` contour. JIVE shows the source code for the current method, and the currently active source code line is highlighted. Different highlighting colors can be used for each thread to further clarify the execution state.

Figure 14 also shows the pop-up menu associated with the contour `DupTree:3`. The user is able to construct views that highlight the specific details of interest by combining different views through this interface. JIVE also supplies convenient commands and settings to alter the view of all contours on the screen. For example, all member tables may be hidden or shown, the view may be expanded, compacted, or minimized, and the default spacing between contours can be altered for each view type.

Call-path and Minimized Views

Figure 3 shows a *call-path view*. The contours with method activations are shown in compact view and those without are in *minimized view*. When minimized, contours are drawn as simple points. A call-path view is convenient when the user wishes to focus on a specific method activation or series of method activations. The overall structure is still visible, but visually complex details are not. If some details were of no interest to the user, he or she could simply hide the entire contour from view through view filtering.

Figure 4 shows a contour diagram that has been fully minimized. The fully minimized view is useful when one wishes to see the overall structure without showing any contour's internal details. We provide convenient actions that will expand all contours, stack all contours, minimize all contours, or show the call-path view. Additionally, the user can select any individual contour and choose to expand, stack, or minimize it, or to show or hide its member table. This puts control over the visualization into the domain of the user. Minimized contours may have their labels hidden; Figure 3 has hidden the labels, and Figure 4 has shown them.

Analysis of Programs and Diagrams

The fact that there is a relationship between diagrams and programs is generally accepted. It is common for those teaching introductory computer science to use graphical notations to demonstrate concepts of object-orientation. The contour model notation provides a standard methodology for visualizing Java runtime states. Using JIVE, we are able to perform a more systematic analysis of the relationship between programs and the diagrams they engender.

Design patterns are used during system design in order to make the best use of the benefits of object-orientation[11]. Analysis of program design is beneficial, but it does not necessarily translate into understanding of runtime states. Design patterns are explained in terms of their design and their runtime

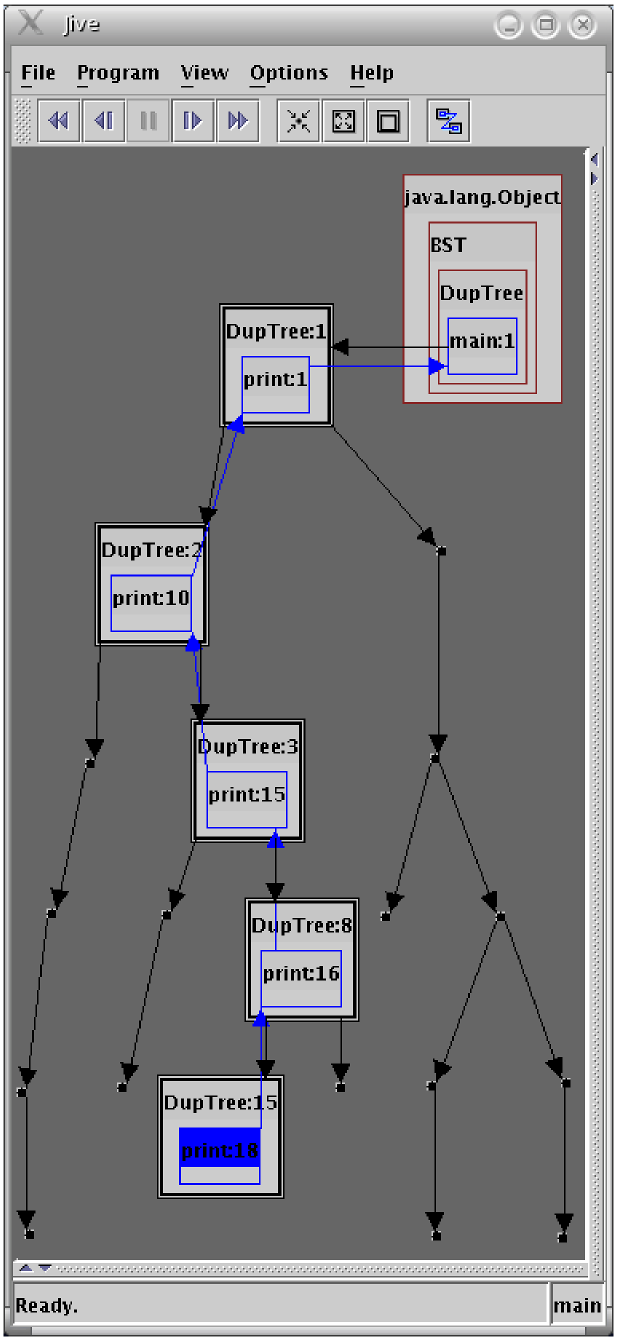

Figure 3: Call-path view of an object diagram. Only those objects that contain active methods in this state are expanded; inactive objects are shown as simple points.

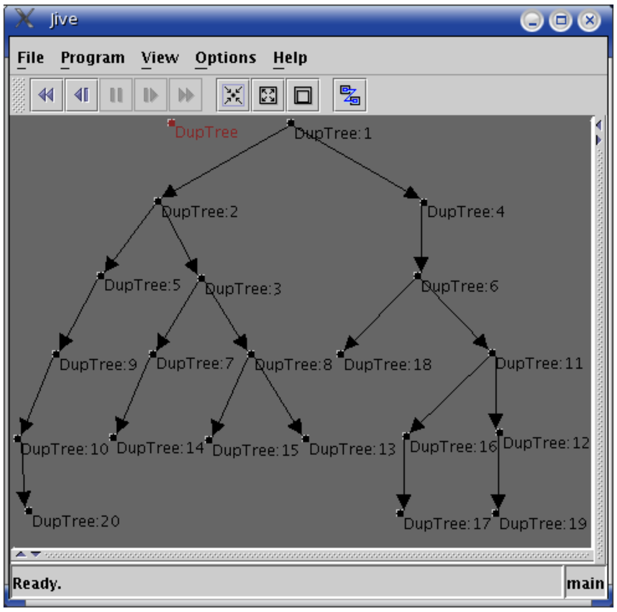

Figure 4: Minimized view of a contour diagram. All objects are shown as simple points.

behavior. Using JIVE, we are able to see and interact with the runtime behaviors, providing an excellent tool for explaining patterns to students; we have used JIVE in this manner in a graduate-level seminar with much success. Furthermore, a user may use JIVE as a visual debugger when analyzing runtime states. The difference between a programmer's mental model of a program and the visualization of the program can expose the location and nature of the error.

Consider as an example an application that uses Java's Swing classes to create a graphical user-interface. The object diagram of such a program will contain a treelike structure of GUI objects, rooted in a window frame and with leaves of buttons, text fields, and other widgets. There should be a small number of observer objects that are connected to the user-interface components and monitor them for state changes. Essentially, the object structure consists of a fixed hierarchy along with instances of the observer design pattern. This structure will repeat in different application domains as long as the programs share a similar GUI structure. It is possible that this type of structure may occur in non-GUI applications as well: consider the object structure of an application that processes a hierarchical filesystem and monitors files for change. The basic structure of these two object diagrams is the same, and so they can be rendered in JIVE in similar ways in order to highlight the parallels.

4 Contour Semantics for Java

The original contour model for object-oriented languages is effective in showing how contour models can be adapted to objects in general[18], but it requires modification in order to be applied to the nuances of specific programming languages. To this end, we have augmented the contour model to produce a visual operational semantics for Java. The specific modifications deal with static contexts, inner classes, multithreaded applications, and overriding and shadowing.

Static Contexts

In the standard object-oriented contour model, runtime states are represented with object and method contours. This model is sufficient for a language where all runtime data is encapsulated in objects and their methods, but Java introduces the notion of *static* members[15]. Variables, methods, and inner classes can be declared as static, and these static members are associated with a class rather than a particular instance of the class. A *static contour* is introduced into the contour diagram for each static context. Since there is only one static contour for any class, no instance count is necessary. Invocation of a static method is represented by nesting a method contour within the appropriate static contour.

We introduce an implicit *static link* that connects an instance contour C_i to the static contour of its defining class, C. Figure 5 shows a small contour diagram where the static links have been drawn explicitly. Drawing these links explicitly for every instance of a class will quickly reduce diagram legibility; it is sufficient to have implicit links since the names of instance contours clearly relate them to their corresponding static contour. Adding a separate static space in the diagram requires that we add the following requirement to the standard search strategy for resolving symbols. Given a class C, a static contour `C` for C, and an instance contour `C:i` of C: if a symbol is not found in the member table of `C:i`, then the static link is followed, and the symbol is sought within the member table of `C`; if the symbol is not found in `C`, then attempt to resolve the symbol in the parent of `C:i`. This implements the proper semantics

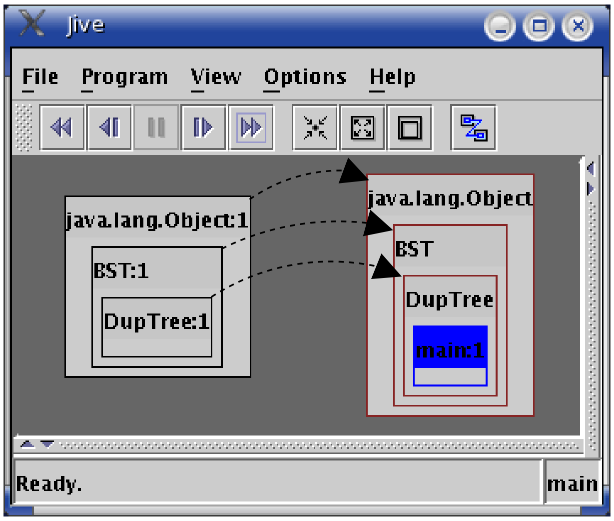

Figure 5: Contour diagram with explicit static links. Static links are usually implicit, and the static contexts are often hidden from view after `main` and other static methods have finished.

of Java's scoping of static members, and it maintains the planarity of the contour diagram.

Inner Classes

An inner class is a class defined within the context of another class[15]. We deal with anonymous inner classes and named inner classes in the same manner, using Java's numbering system to identify anonymous inner classes. Non-static inner class instances are always contained within exactly one enclosing instance, and so to reflect this, the instance contour for the inner object is nested within the instance contour of the enclosing instance. Non-static inner classes may not define static members, and so unlike non-inner classes, there is no static counterpart to a non-static inner class. Static inner classes are associated with a class' static context, and so the instance contours for static inner classes are nested within the static contour for their enclosing class. Static inner classes may define static members, and so each static inner class has a static contour; the static contours of static inner classes require no special handling and are treated as any other static contour.

As an example, consider an alternate binary search tree implementation that uses inner classes, a partial implementation of which is given in Figure 6. The `BST2` class defines a non-static inner class, `Node`, and the tree structure is recursively defined over `Node` instances rather than `BST2` instances. Figure 7 shows the state of the simple program when the second tree node has been inserted. The two instances of the inner class, `BST2$Node:1` and `BST2$Node:2`, are both nested within the single instance of `BST2`, `BST2:1`. This figure also serves to illustrate the problem of nested graph processing in contour diagrams: any contour may have an arbitrarily complex and dynamic nested contour structure.

```
public class BST2 {
  private Node root;
  public void BST2() {}
  public void insert(int data) {
     if (root==null) root = new Node(data);
     else root.insert(data);
  }
  protected class Node {
     int data; Node left, right;
     Node(int data) { this.data = data; }
     void insert(int data) {
       if (data < this.data)
         if (left==null)
   left = createNode(data);
         else left.insert(data);
       else if (data > this.data)
         if (right==null)
   right = createNode(data);
         else right.insert(data);
     }
  }
  ...
}
```

Figure 6: Alternate binary search tree implementation using inner classes.

Overriding and Shadowing

The traditional contour model for object-oriented languages adds two *intracontour links* to the member table of each instance contour: a `this` (or `self`) link that points to the innermost object of a stack of contours, and a `super` pointer that points to the superclass contour of a contour, if one exists[18]. The intention of the `this` intracontour link is to show overriding; an expression such as `this.toString()` will always refer to the implementation of `toString()` that is deepest in the class hierarchy. However, due to Java's variable shadowing and method overriding, the meaning of the `this` link can be deceiving regardless of how it is drawn. Drawing the `this` link as described creates a contradiction when referencing shadowed variables through it. Similarly, if `this` refers only to the contour containing itself, which would provide a clear visual semantics for shadowing, then the overriding of methods is not clearly expressed. This is an inherent shortcoming of any visual operational semantics for Java. It is impossible to resolve the method and variable referencing uses of `this` using only one intracontour link due to its two syntactic uses.

The complications of `this` are not insurmountable in JIVE. The dichotomy between method overriding and variable shadowing can still be explained using JIVE diagrams since the different contour contexts are clearly visible in a detailed view. Since object-orientation favors encapsulation of data, we expect method overriding to be much more prevalent than variable shadowing in real Java applications. Therefore, in JIVE, we do include a `this` reference in the member table which refers to the innermost instance contour in a collection of nested contours, and we leave the detailed explanation of overriding and shadowing to an instructor.

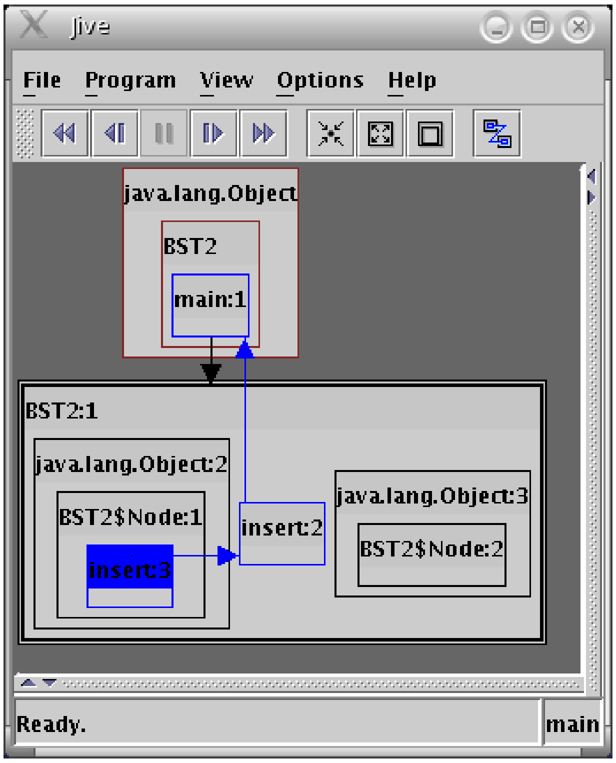

Figure 7: Contour diagram showing non-static inner class instances. Each non-static inner object is contained within its enclosing instance.

Threads

Java supports multithreaded applications; in fact, every Java program that has a Swing or AWT user-interface is inherently multithreaded since the JVM will automatically start the AWT-Event thread to process user input. Multiple concurrent threads imply multiple simultaneous paths of method calls. This is easily represented in the contour model through visual cues on methods and their return links. Each thread's path of execution is drawn in a different color. Even if the same method definition is being used by multiple threads, each thread has its own method contour since each thread has its own stack. The same colors are used to highlight the multiple threads of a contour diagram and sequence diagram as well as the source code; this enforces the interdependence of the view of the current state (contour diagram and source code highlighting) and the history of execution (sequence diagram). The JIVE-generated sequence diagram for a simple multi-threaded program is shown in Figure 8. The program being visualized simply starts two threads and lets them race to a finishing condition.

JIVE is able to visualize programs that have their own graphical user-interfaces. Such applications

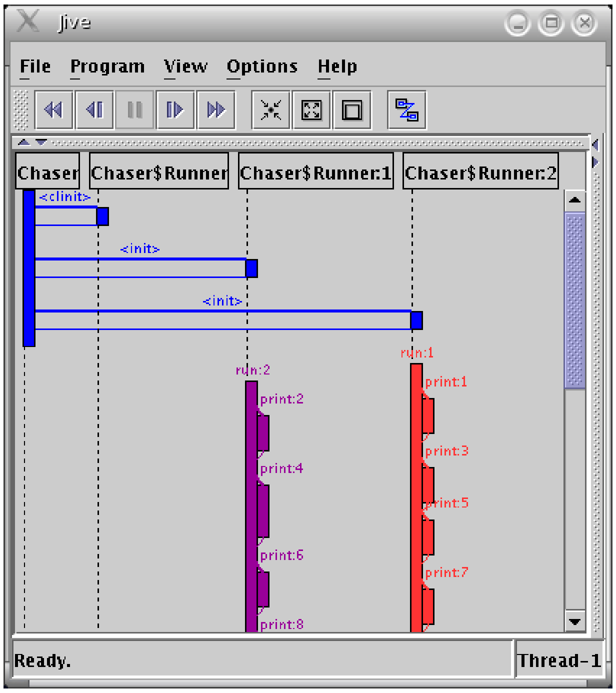

Figure 8: Sequence diagram with concurrent threads. The main thread, shown in blue, terminates after initializing the two `Chase$Runner` thread objects. The other two threads have set up a simple chase condition.

are loaded into JIVE like any other program, and the user-interface for the application co-exists with JIVE's interface. The next section describes how JIVE's two-process architecture makes this possible.

5 JIVE Architecture

We have incorporated our model for understanding program execution into the prototypical tool JIVE [13]. An overview of the architecture is shown in Figure 9. In this section, we describe the design and interaction of these components.

Two-Process Architecture

We have explored the use of source-code transformation in order to produce visual representations of the runtime state in previous visualization tools[12]. This model is difficult to maintain for a growing language such as Java. Each time the language or libraries change, changes must be made to any custom compilers or interpreters that are written.

Our current approach abandons program transformation in favor of a two-process architecture. The visualization environment itself runs in one process. The user provides to JIVE the program he or she wishes to visualize, and JIVE starts the application in a second process, called the *client* process. Communication between the two processes is made possible by the Java Platform Debugger Architecture (JPDA)[28]. In order to guarantee source-code highlighting functionality, a program must be loaded from its source code, but JIVE can run visualizations from compiled class files as long as they contain debug information. JIVE supports multithreaded programs that are uniprocessing, but the design does not currently allow for visualization of distributed or multiprocessing applications.

Figure 10 shows a high-level sequence diagram that depicts the interactions between JIVE and the client process. Once JIVE has started the client process, it registers event listeners with it via JPDA and awaits notification. When the client's state changes, its execution is suspended, and notification of the event is sent to JIVE for processing. Once the data model and the appropriate views have been updated, JIVE resumes the client program and returns to waiting for events.

The amount of processing that the client performs before suspending is the *step size*. JIVE allows for different granularity of step sizes, including individual source code lines, method invocations, or arbitrary breakpoints in the source code. It is also possible to disable event suspension, in which case events stream continuously into JIVE, which processes them sequentially.

We have tested JIVE on programs written by students in introductory programming classes, and we have anecdotal evidence that the performance impact of JIVE is low in these cases when executed on modern desktop computers. The longest delays occur during the initialization of the AWT or Swing libraries in applications with graphical user interfaces. More quantitative analysis of JIVE's performance impact is planned as future work (see Section 6). .

Interactive Execution

As the client program runs, it is monitored for changes. The changes are recorded into the Execution Log module, shown in Figure 9. The log is coupled with a database into which execution history is stored. The JIVE prototype uses an in-memory model for faster queries and processing, but the en-

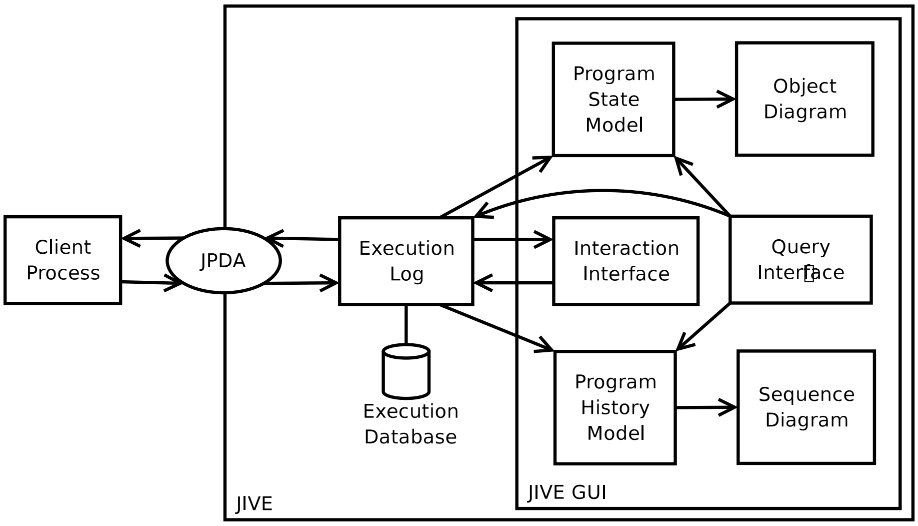

Figure 9: JIVE architecture. Boxes represents JIVE modules, and arrows represent data and control flow.

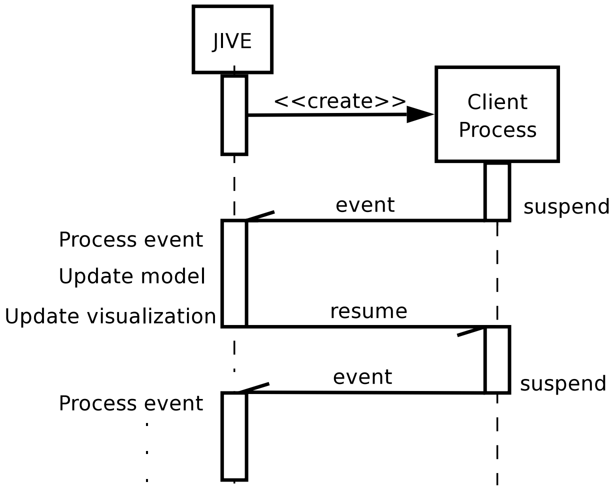

Figure 10: Overview of the interaction between JIVE and the visualization client. JIVE and the client run on two separate processes, shown as lifelines in the sequence diagram. We are able to provide interactive forward execution using the JPDA suspension and breakpoint functionality.

tire model can be externalized in situations with excessive data or limited memory. Additionally, the log can be serialized for offline analysis.

JPDA events received from the client are interpreted into a simpler set of events by JIVE. The execution history is therefore expressed through a sequence of declarative events [26]. These events are memento objects which are able to commit or uncommit themselves from a program state model [11]. In this way, multiple states can be shown at once by notifying the state models which events have been committed or rolled back. The events used by JIVE reflect the following execution events:

- static context creation
- object creation
- method call
- method return
- exception thrown and caught
- change in source line
- change in variable value

Each of these events is encapsulated as an object, and the object contains enough information about the event to commit or un-commit itself from the execution log. Essentially, each event contains a prototype of the change it will make to the execution state, in a manner similar to templates or prototype-driven programming [30]. An example interaction log from the first few steps of the DupTree example are shown in Figure 11. The events are represented textually in the figure, although in practice they are either in-memory objects or the XML serializations thereof.

1. Create `Object` static context.
2. Create `BST` static context in `Object`.
3. Create `DupTree` static context in `BST`.
4. Call `main:1` in `DupTree` on thread `main`.
5. Highlight line 1 in `main:1`.
6. Create `Object:1` object.
7. Create `BST:1` object in `Object:1`.
8. Create `DupTree:1` object in `BST:1`.
9. Call `<init>` in `DupTree:1`.
10. Highlight line 1 in `<init>` on thread `main`.
11. *etc.*

Figure 11: Sample execution log for the `DupTree` application, displayed as descriptive text. This log shows the first two *steps* of the execution: the first includes events 1–5, and the second includes 6–10.

Stepping backward in JIVE does not affect the client program, but rather it only affects the visualization. The client program is suspended while the user inspects past states, and it is resumed when necessary. While this model does preclude the possibility of altering program inputs, it also avoids the problem of maintaining synchronization with external resources such as data streams or input/output devices. For example, when a program with a graphical user-interface is run through JIVE and the user steps backward, the interface will cease responding while the client process is suspended. This is because the client's event thread, which is responsible for redrawing the client program's interface and processing input events, is suspended along with all other threads. If a program were un-executed (as opposed to rewinding visualizations), irreversible computations may be performed, or a stream may be read that cannot be pushed back; hence, we abide by a general model that client programs are unilaterally suspended while visualizing recorded states.

Multithreaded applications are handled elegantly in our execution model. Since we restrict to uniprocessing systems, there is only ever one instruction being executed at a time. JIVE is aware the thread on which an event occurs, as shown in Figure 11. The active thread is recorded in the execution log and reflected in the visualization (see Figure 8).

Drawing Contour Diagrams

Contour diagrams are complicated structures, and their drawing is a difficult task. There is a wide range of research into dynamic graph algorithms[10], but it is not clear that contour diagrams can be processed as simple dynamic graphs. The specific properties of contour diagrams that complicate automated graph drawing are:

- *Nested structures.* Within a contour there may be an complex nested diagram.

- *Multiple types of nodes.* There are instance contours, static contours, and method contours, and each behaves differently. Additionally, special representations may be required for visualization of certain Java types such as arrays.

- *Multiple types of edges.* There are structural links and return links, and they have different behaviors. Method return links are tied to the life-cycle of a single method contour. Structural links from method contexts have substantially different lifetimes than structural links from object members.

- *Multiple types of crossings.* A generally accepted graph-drawing aesthetic is that the crossings of edges should be minimized [10]. In contour diagrams, there can be crossings between heterogeneous links and between the boundaries of different types of contours.

It is possible to convert contour diagrams into multigraphs. Different weights can be given to different types of links. In our experimentation, structural links have been more permanent than method return links, and so we give structural links a higher weight in the conversion to a multigraph (that is, more edges are created in the graph for a structural link than for a return link). The resulting graph is then layered using a modification of Coffman-Graham layering[6, 10] for multigraphs, and the crossings are minimized, yielding a layered graph drawing. An inverse transformation is applied to revert the multigraph into the contour diagram. In our preliminary testing, this works well for simple hierarchical structures such as trees and graphical user-interface composition. Simple greedy layouts, where objects are simply drawn in the order they are created and locked into their positions, have proven generally ineffective in our experimentation.

The goal of JIVE is to provide a robust environment for visualization, and view filtering with multiple concurrent views gives the user the freedom to focus on details of interest. When using JIVE as a teaching tool, it may be useful to show fully-detailed contour diagrams to explain concepts such as inheritance and aggregation. However, for larger programs, it is impractical to show this much detail. JIVE supports multiple views with multiple levels of detail. We also have a model for filtering the visualizations; for example, a user may only be interested in their own classes, so they may apply a filter to exclude the visualization of all classes in packages starting with "java". Another user may wish to study the Swing packages, so he or she could exclude all "java" classes except those in the "javax.swing" package and subpackages. Default sets of filters can be used for general application in pedagogic or debugging applications; for example, an instructor wishing to use JIVE could configure filters known to be useful for the specific applications being visualized.

Runtime History Database

The interaction model maintains a complete traversable record of program history. These storage structures can be conceptualized as an object-oriented database of runtime state information. In a relational database, queries are performed on a table or groups of tables, and the results of these queries can be values and new tables. In our runtime state database, queries are performed on the history of runtime execution or on portions thereof, and the results are values, sets of states, or portions of program history. Furthermore, we have a visual representation of program states and program history, and so both queries and the results can be visual.

One aspect of runtime database querying is *variable tracking*. A variable is selected in JIVE, and its value is monitored for changes. An instance or local variable may be tracked in a particular object or across all instances of an object. Static variables are tracked in the static context and for usage in objects of the class. When the user initializes variable tracking, he or she specifies a condition for the variable, such as its value being changed or within a range (for numerical values). The result of the variable tracking is highlighted through the JIVE interface. As an example, consider variable tracking of the `count` variable in the `DupTree` class from the previous examples. A user investigating the `DupTree` may wish to know when the count changes to a value besides its default of 1. The result of this query is shown in Figure 12. In the visualization of the current state, `DupTree:2` is highlighted since it is the contour that contains the `count` member that changed. Additionally, the `insert` method that caused the change in `count` is highlighted in the sequence diagram, and an arrow points to the specific point at which the change was made.

JIVE's architecture allows for the execution history database to be written to a file for future use. A program's history file can be loaded and visualized as if the program was running, but without needing to actually compile and run the program. This serialization of execution histories also makes it possible to compare executions. Multiple execution histories may exist for a program as small changes were made to it, and through JIVE, a user can visualize the differences between program executions. It is also possible to perform queries that span program executions, and the results of these queries are sets of views.

6 Conclusions and Future Work

Our preliminary studies of JIVE's effectiveness have been very positive. We have used different forms of the tool and the visual semantics in introductory-level computer science courses, intermediate undergraduate courses, and graduate-level courses. Object diagrams have proven useful for debugging, especially in cases where the user has a mental map of what structure is desired, and JIVE clarifies states are actually produced by the erroneous program. The JIVE-generated sequence diagrams have proven invaluable in explaining the behavior of various design patterns and program constructs, and they are further clarified by the capability to view the details of the object and method calling structure at varied levels of detail. The color-highlighting of threads in object and sequence diagrams has been instrumental in explaining the behavior of multithreaded programs. This has been especially true for programs with graphical user-interfaces: the sequence diagram clarifies how the event-processing thread handles user input and drawing the GUI, and how complicated processing tasks can be deferred to other threads.

Though the sequence diagrams have proven highly effective in both explaining program execution and detecting errors, program history becomes very complicated in large or long-running programs. The stan-

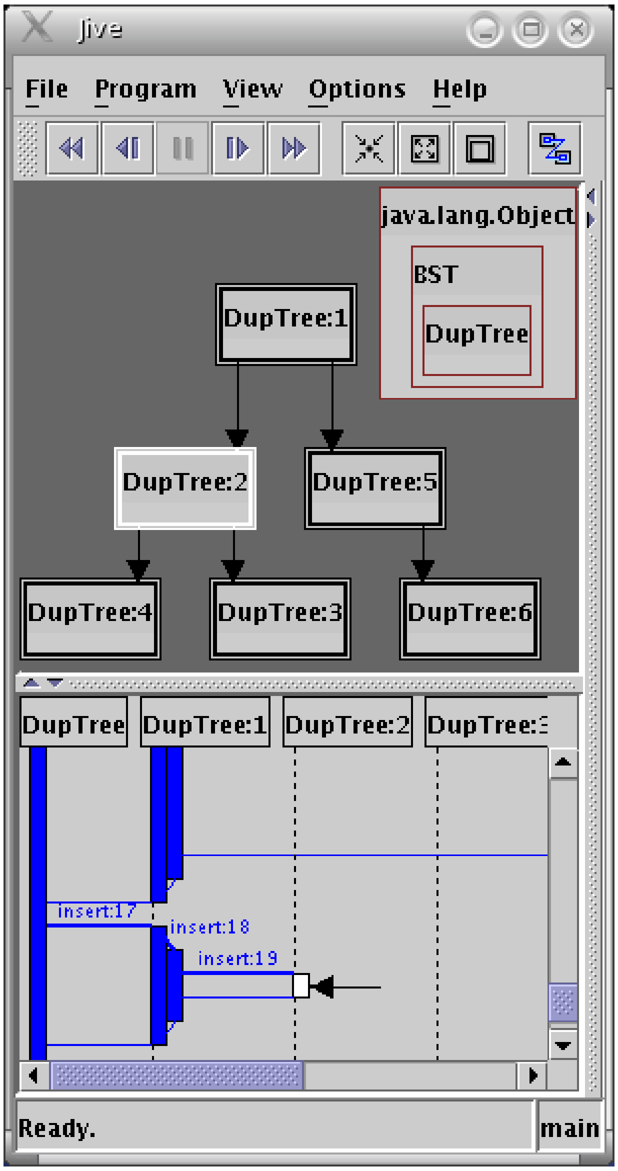

Figure 12: Result of a runtime history query, specifically a query about the value of the data field in any `DupTree` instance. The sequence diagram highlights *when* the query is true, and the object diagram highlights *where* the query is true (in white).

dard use of sequence diagrams is for system design, when there is an inherent level of abstraction. We are investigating means by which the program history information can be modularized and multiple views can be provided akin to the compact, minimized, and detailed views of contour diagrams. In addition to the aesthetic problem of providing a comprehensible diagram, there is a theoretical problem of how to most efficiently structure a dynamically-growing sequence diagram. This is an area where analysis of program source code may provide an insight into the optimal ordering of lifelines before program execution even begins.

The current interaction module supports only stepping forward and backward or running to breakpoints, including the start and end of a program as default breakpoints. This model makes it inefficient to "jump" from one program state to another distant state since all of the changes between the two states must be processed sequentially. Despite this inefficiency in processing time, it should be noted that the stated approach is optimally efficient in storage space, since each change that must be recorded is recorded exactly once. However, our studies confirm that shorter response-times provide for a better atmosphere for understanding a program. The best way to increase the effectiveness of these jumps between states is to store redundant, composite transaction information at key points in program execution. The source code, when it is available, can be analyzed in order to determine the dependencies among methods and data[5, 16]. We are currently developing an ontology of programs in order to determine the analysis techniques that produce the best results for different types of programs.

Generating good drawings of object diagrams is a difficult problem. Despite the wide range of graph-drawing techniques, the unique properties of our diagrams prevent the direct application of known techniques. Existing dynamic graph drawing approaches[23, 22, 7, 27] handle the problem of general dynamic graphs, but it is not clear that diagrams of object runtime interactions can be constructed as simple graphs. We are investigating how to apply program analysis techniques such as call-graph generation[16] and object-oriented program slicing [5, 21] to extract program-specific aeshetics to use in addition to graph-theoretic aesthetics described earlier.

We are continuing to improve our understanding of runtime history as a queryable database. Our current set of queries supported by our model is limited, but new aspects of runtime queries continue to emerge. We are currently developing a more formal query language, improving result organization, and enhancing the visual interfaces. The runtime history database also has many applications outside of program visualization and comprehension. Comparative analysis of program execution is a useful tool for system security testing; by running a program many times, it may be possible to extract a mathematical description of what "normal" execution is. If a program then exhibits aberrant behavior, it can be marked as a potential security risk. Additionally, the results of the comparison can be visualized with JIVE.

Acknowledgements

Ashim Garg helped develop the theory behind JIVE's dynamic drawing capabilities. The internal representation of states was developed with the help of Eric Crahen. Hani Girgis, Akshay-Venkataram Hegde, and Manu Pushpendran have assisted in the development of JIVE's query system and its graphical interface.

References

[1] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23(6):589–616, 1993.

[2] R. M. Balzer. Exdams: Extendable debugging and monitoring system. In *Proc. AFIPS*, volume 34, pages 567–580, Montvale, N.J., 1969. AFIPS Press.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley Longman, Inc., Reading, Massachusetts, 1999.

[4] M. H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Software*, pages 28–39, January 1985.

[5] Z. Chen and B. Xu. Slicing object-oriented java programs. *SIGPLAN Not.*, 36(4):33–40, 2001.

[6] E. G. Coffman, Jr. and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

[7] R. F. Cohen, G. Di Battista, R. Tamassia, I. G. Tollis, and P. Bertolazzi. A framework for dynamic graph drawing. In *Proceedings of the eighth annual symposium on Computational geometry*, pages 261–270. ACM Press, 1992.

[8] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 326–337. ACM Press, 1993.

[9] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan. Drive-by analysis of running programs. Technical report, IBM T.J. Watson Research Center, 2001.

[10] P. Eades, G. Di Battista, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualisation of Graphs.* Prentice-Hall, New Jersey, 1999.

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

[12] P. Gestwicki and B. Jayaraman. Interactive visualization of Java programs. In *IEEE Symposium on Human-Centric Computing, Languages, and Environments*, pages 226–235, September 2002.

[13] P. Gestwicki and B. Jayaraman. Jive: Java interactive visualization environment. In *OOPSLA 2004 Conference Companion*, pages 615–616, 2004.

[14] A. Goldberg and D. Robson. *Smalltalk – The Interactive Programming Environment.* Addison-Wesley, 1980.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition.* Addison Wesley Longman, Inc., Reading, Massachusetts, 2000.

[16] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 108–124. ACM Press, 1997.

[17] T. D. Hendrix, J. H. Cross, II, and L. A. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight ide. In *Proceedings of the thirty-fifth SIGCSE Technical Symposium on Computer Science Engineering*, March 2004.

[18] B. Jayaraman and C. Baltus. Visualizing program execution. In *IEEE Symposium on Visual Languages*, pages 30–37, September 1996.

[19] J. B. Johnston. The contour model of block-structured processes. *ACM SIGPLAN Notices*, pages 255–282, 1971.

[20] M. Kölling and J. Rosenberg. Guidelines for teaching object orientation with java. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 33–36. ACM Press, 2001.

[21] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th international conference on Software engineering*, pages 495–505. IEEE Computer Society, 1996.

[22] S. Moen. Drawing dynamic trees. *IEEE Softw.*, 7(4):21–28, 1990.

[23] S. C. North. Incremental layout in dynadag. In *Proceedings of the Symposium on Graph Drawing*, pages 409–418. Springer-Verlag, 1996.

[24] A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. In *Australian Software Engineering Conference*, pages 251–261, 2004.

[25] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 13–22. IEEE, 1999.

[26] T. Richner, S. Ducasse, and R. Wuyts. Understanding object-oriented programs with declarative event analysis. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*. Springer-Verlag, July 1998.

[27] K. Ryall, J. Marks, and S. Shieber. An interactive constraint-based system for drawing graphs. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 97–104. ACM Press, 1997.

[28] Sun Microsystems, Inc. *Java Platform Debugger Architecture Overview*, 2001.

[29] Sun Microsystems, Inc. *Java 2 SDK, Standard Edition, Version 1.5.0 Summary of New Features and Enhancements*, 2004. In the Java 2 SDK 1.5 documentation.

[30] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM Press, 1987.

[31] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 271–283. ACM Press, 1998.

[32] M. V. Zelkowitz. Reversible execution. *Commun. ACM*, 16(9):566, 1973.

[33] A. Zeller and D. Lutkehaus. DDD - a free graphical front-end for UNIX debuggers. *SIGPLAN Notices*, 31(1):22–27, 1996.

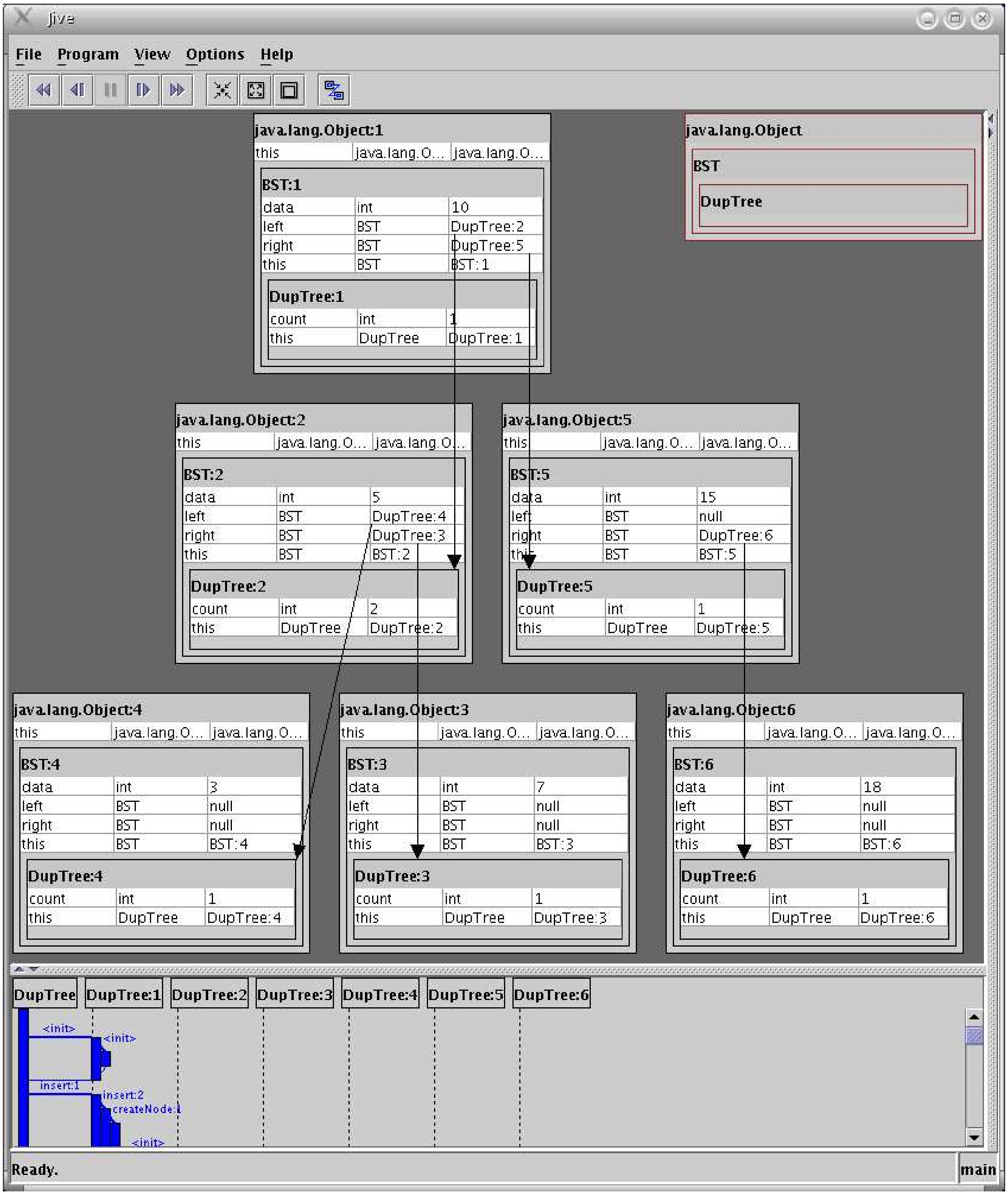

Figure 13: Detailed view of a binary search tree with all member tables shown. This view shows all inherited variables; for example, it is clear that each `DupTree` object is inheriting `data`, `left`, and `right` from `BST`. These tables can be expanded to show the methods that are declared in each context as well.

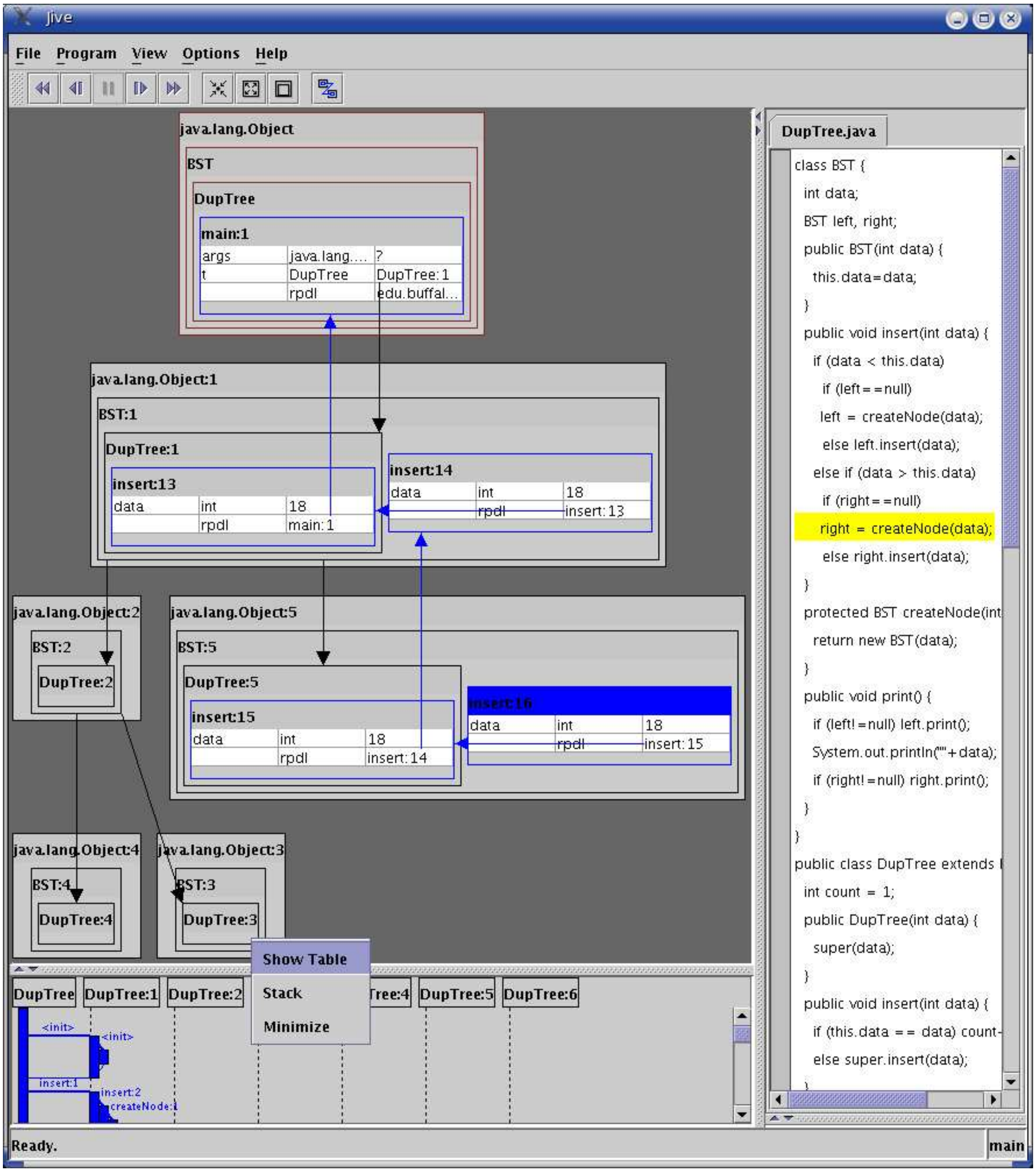

Figure 14: Contour diagram with method activations and source code highlighting. The highlighted code is updated on each program step, forward or backward. Also visible is a pop-up menu of options to change the view of instance contour `DupTree:3` to show its table, stack it (convert to compact view), or minimize it.