Interactive Visualization of Object-Oriented Programs

by

Paul V. Gestwicki
June 15, 2005

A dissertation submitted to
the Faculty of the Graduate School of
University at Buffalo
The State University of New York
in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy

Department of Computer Science and Engineering

Acknowledgements

There are many people whose guidance and inspiration have helped me to complete this dissertation. I am grateful to the faculty and staff of the University at Buffalo Department of Computer Science and Engineering, especially my dissertation committee of Dr. Bharat Jayaraman, Dr. Ashim Garg, and Dr. Bina Ramamurthy, as well as Dr. Li Lin of the Department of Industrial Engineering. This work would not have been possible without the vision and support of my adviser, Dr. Jayaraman, and I am grateful for his years of encouragement and collaboration. I would also like to thank Dr. William Rapaport and Dr. Helene Kershner for their inspiration and assistance during my stay at the University at Buffalo. Several graduate students have contributed to the JIVE methodology, its implementation, and our experimentation with the system; specific thanks are due to Eric Crahen, Hani Girgis, Akshay Hegde, Blake Martin, Manu Pushpendran, Shalaka Sindkar, and Li Xu. I am grateful to my friends and my family for their years of encouragement. Finally, I wish to thank to my wife, Jessica, who has been a source of support throughout my graduate studies, and to whom I dedicate this dissertation.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

We present a novel approach for the interactive visualization of the execution of object-oriented programs that fulfills several important criteria: (i) visual depiction of the current execution state as well as the entire history of execution; (ii) support for forward as well as reverse execution stepping; (iii) enhanced graph-drawing techniques for the current state and history of execution; (iv) support for run-time queries on the execution state; (v) support for all major features of the Java language and use of the standard Java compiler and run-time system. Taken together, our approach marks a significant advance over existing techniques for visualizing object-oriented program execution.

Our proposed notation for expressing Java execution states clarifies the important fact that objects are environments; that is, a method execution is shown within its object context. By incorporating diagrams similar to those used during the design phase, we help to close the loop between design and execution. In particular, an enhanced UML-like object diagram is used for depicting the current state, and sequence diagrams are used for execution history. Interactive execution is instrumented by conceptualizing program execution as a database, so that stepping backward or forward through recorded states involves rolling back or re-committing transactions. Our methodology incorporates a novel technique for recording minimal state transitions in order to efficiently provide this interactive execution.

Automatically generating object and sequence diagrams requires advanced graph drawing techniques. The enhanced object diagrams we present are not simple graphs, and special techniques are required to process them. We illustrate how layered drawing techniques can be used to achieve good drawings of enhanced object diagrams. Two techniques are presented for automatically drawing sequence diagrams: a stochastic simulated annealing approach and a fast greedy technique. We show how traditional graph-theoretic approaches can be combined with program-specific properties in order to produce better automatic drawings. Specifically, an analysis of the class diagram is used to determine the classes that form logical clusters, and these clusters are formed in the object diagram generated at runtime. We have implemented our approach in a prototypical tool called JIVE: Java Interactive Visualization Environment.

Chapter 1

Introduction

As computers have become more powerful, programming languages and programming environments have become increasingly more complex. The seeds of the modern software development environment can be found in projects such as Ivan Sutherland's *Sketchpad*, regarded by many as the first object-oriented programming environment [87]. Now there is a proliferation of widely-used integrated development environments, including Microsoft's VisualStudio, the open-source prodigy Eclipse, IBM's Rational Rose, and Borland's Builder and Together tools. However, as programming environments and software development techniques have become more advanced, there has not been a corresponding growth in tools to understand program behavior at runtime. The majority of debuggers that come with integrated development environments are graphical wrappers around traditional text-based debuggers such as `dbx` [65]. Despite efforts to advance debugging to the same interactive, graphical plane as development and testing tools [*e.g.* 10, 69, 51], the adoption of such tools has been significantly lower.

The widespread acceptance of object-oriented design and programming languages has accompanied the advancement of the state of the art. This is a natural evolution: as systems have become more complex, there is greater need for encapsulation, reusability, and modularity. These features are all granted by proper application of an object-oriented methodology. Many argue that *design patterns* are the foundation of good object-oriented programming [15]. In fact, these ideas are so fundamental to software engineering that they can be applied without using an "object-oriented" language at all, though language-level support yields cleaner syntax. Recent years have seen a shift in computer science pedagogy towards the use of Java, an object-oriented programming language [41]. The name *Java* refers to a programming language, a virtual machine specification, and an application programming interface (API). Patterns and object-orientation have become the cutting edge of computer science pedagogy. However, contemporary programmers and students of computer science still suffer from the disparity between development tools and program understanding tools.

Object-oriented programming differs from traditional procedural programming in two important ways. First, objects are environments of program execution; that is, a method execution can only be understood within its surrounding context. Second, method calls frequently result in context switching among objects. Object-oriented design encourages the use of many small methods — accessors and mutators for example — and so these context switches tend to occur often. This is an artifact of object-oriented abstraction, an important methodology that mandates a black-box approach, wherein the state of an object is hidden behind a public interface. Unfortunately, such abstractions can lead to further disparity between understanding a design and understanding its implementation due to the separation of concerns. Effective debugging requires an understanding of both levels [9].

The goal of this dissertation is to help bridge the gap between development and debugging tools. The technique presented herein is a novel approach to interactive execution of Java programs, and it can be used to enhance program understanding and hence facilitate debugging. The focus is on Java since it has seen widespread use throughout academia and industry, and its virtual machine architecture facilitates implementation of our methodology. The remainder of this introductory chapter is structured as follows:

- Section 1.1 presents an example that motivates the need for the research and techniques described in this dissertation.
- Section 1.2 presents our seven desiderata for effective interactive visualization of object-oriented programs.
- Section 1.3 provides an outline of our approach and the technical results achieved.
- Section 1.4 gives an outline and overview of the remainder of this dissertation.

1.1 Motivating Example

Consider a simple binary search tree, a familiar data structure studied by undergraduate computer science students. A sample Java implementation of a binary search tree of integers is provided as Listing 1.1. This example is deceivingly simple to one who is new to object-orientation. The `insert` method is recursive in that it calls itself, but the recursive calls are made in different object contexts. Understanding the meaning of expressions such as “`value < data`” requires an understanding of

```java
public class BST {
  private BST left, right;
  private int data;
  public BST(int value) {
    this.data = value;
  }
  public void insert(int value) {
    if (value < data)
      if (left==null) left = new BST(value);
      else left.insert(value);
    else if (value > data)
      if (right==null) right = new BST(value);
      else right.insert(value);
  }
  public static void main(String[] args) {
    BST root = new BST(100);
    root.insert(75);
    root.insert(125);
    root.insert(50);
    root.insert(80);
  }
}
```

Listing 1.1: Binary search tree with integer data in Java

the scope of both variables. One of them, `value`, is a formal parameter of the method; the other, `data`, is a field of the object. The meaning of `data` changes depending on the actual object context in which the method is called.

Figure 1.1 provides a screenshot of the Eclipse[1] debugger being used on the `BST` application. The Eclipse debugger is a standard text-based debugger similar to those found in both commercial and free integrated development environments. The state shown in Figure 1.1 is an invocation of the `insert` method with the value 80. The debugger is capable of showing the complete state of the program; however, the overall object structure is not shown. It is possible to infer the object structure through the repeated setting of breakpoints, stepping into and out of method activations, and searching through the "Variables" pane, which Eclipse uses to show objects in a tree view; however, this is clearly cumbersome. A related problem is that, without the capacity for reverse execution, users frequently step over the state in which they were interested [1]. Since runtime errors can only be detected after they have occurred, it is easy to step through a state, only realizing it was interesting afterward, and hence requiring a complete re-execution of the program in order to effectively step backward one unit.

A more graphical approach can be taken in order to address the deficiencies of text-based debug-

[1] `http://www.eclipse.org`

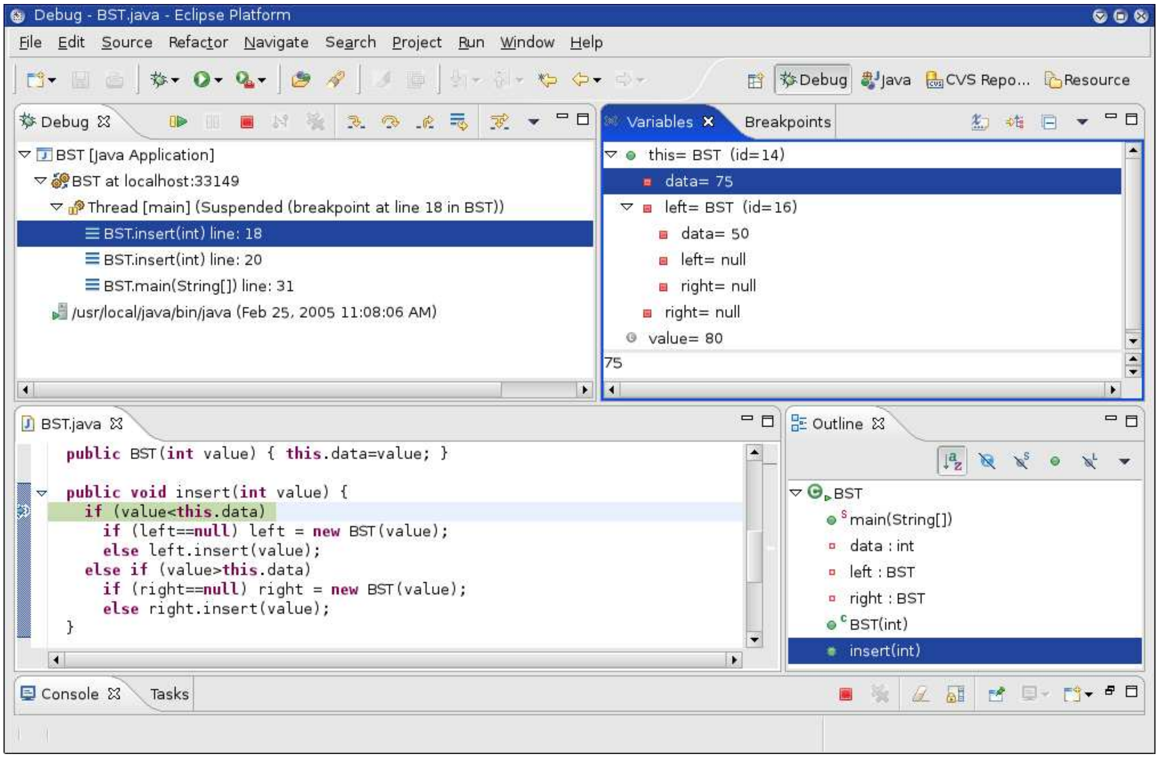


Figure 1.1: The Eclipse debugger being used on the BST application. While the contents of the current object (`this`) is visible through the *Variables* panel in the upper-right, the overall object structure must be inferred by the user by stepping through contexts.

gers. Figure 1.2 shows an object diagram in the Unified Modeling Language (UML) [15] depicting the same state as the debugger of Figure 1.1. The object structure is explicit in Figure 1.2, and the roles of the objects indicate the fields through which they are referenced. Furthermore, the diagram has been manually arranged to be easily recognizable as a tree, with the `left` nodes on the left and the `right` nodes on the right. However, missing from this figure is any notion of the state of method activations. Since methods are not shown, there is no way to clarify the precise meanings of `value` and of `data`. In defense of the UML, it is inherently a *modeling language* and is therefore intended for use during the design phase. However, it is desirable that depictions of runtime state, through a debugger interface, use similar representations to those used during design.

1.2 Desiderata for an Interactive Visualization Environment

A successful visualization environment should combine the flexibility of a traditional debugger with the graphical expressiveness of notations like the UML. We have identified the following seven desiderata for a system that visualizes Java execution:

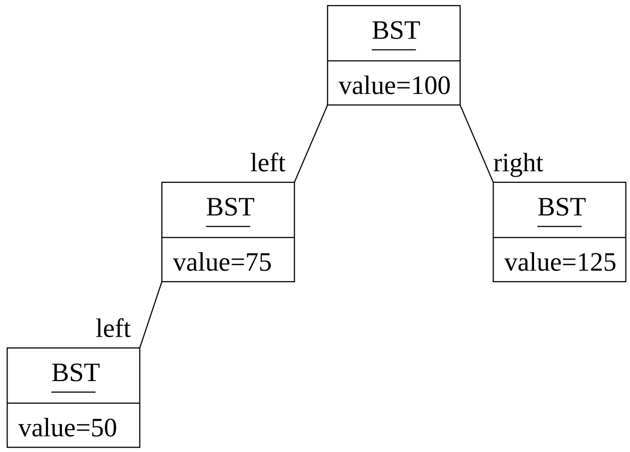


Figure 1.2: UML object diagram depicting a state of the binary search tree application

1. *Depict Objects as Environments.* As noted earlier, the execution states of object-oriented programs differ fundamentally from those of procedural programs since an object is an environment within which method activations take place. There are several tools that clearly depict method call sequences and 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 comprehensibility. The system should allow a user to view abstract, simplified relationships or to examine specific details of an inheritance hierarchy or complex data structure, for example. This flexibility allows the visualizations to be useful for both teaching and debugging, and it facilitates use by those with varied levels of experience.

3. *Capture History of Execution and Method Interaction.* The history of program execution should be observable using notations such as the time-sequence or collaboration diagrams of the UML. 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 enforces the perspective of a queryable database of runtime states.

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. *Use Existing Java Technologies.* It is important for the visualization system to run 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 syntactic enhancements of Java 2 Standard Edition 5.0, which support simpler iteration, generics, and enumerated types, would require changes to any custom compilers and interpreter. Additionally, it should be possible to visualize programs with graphical user-interfaces built from libraries such as Swing and AWT.

These criteria have guided our research into effective means of visualizing execution states and runtime details. There are many tools that have addressed related aspects, many of which are surveyed in Chapter 2, but none satisfy the core criteria that we have formalized. 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 this end, we have created a tool called *JIVE: Java Interactive Visualization Environment*. This dissertation describes the *JIVE methodology*: the motivation, research, design, implementation, and application of the JIVE tool. Figure 1.2 is a screenshot of the JIVE tool,

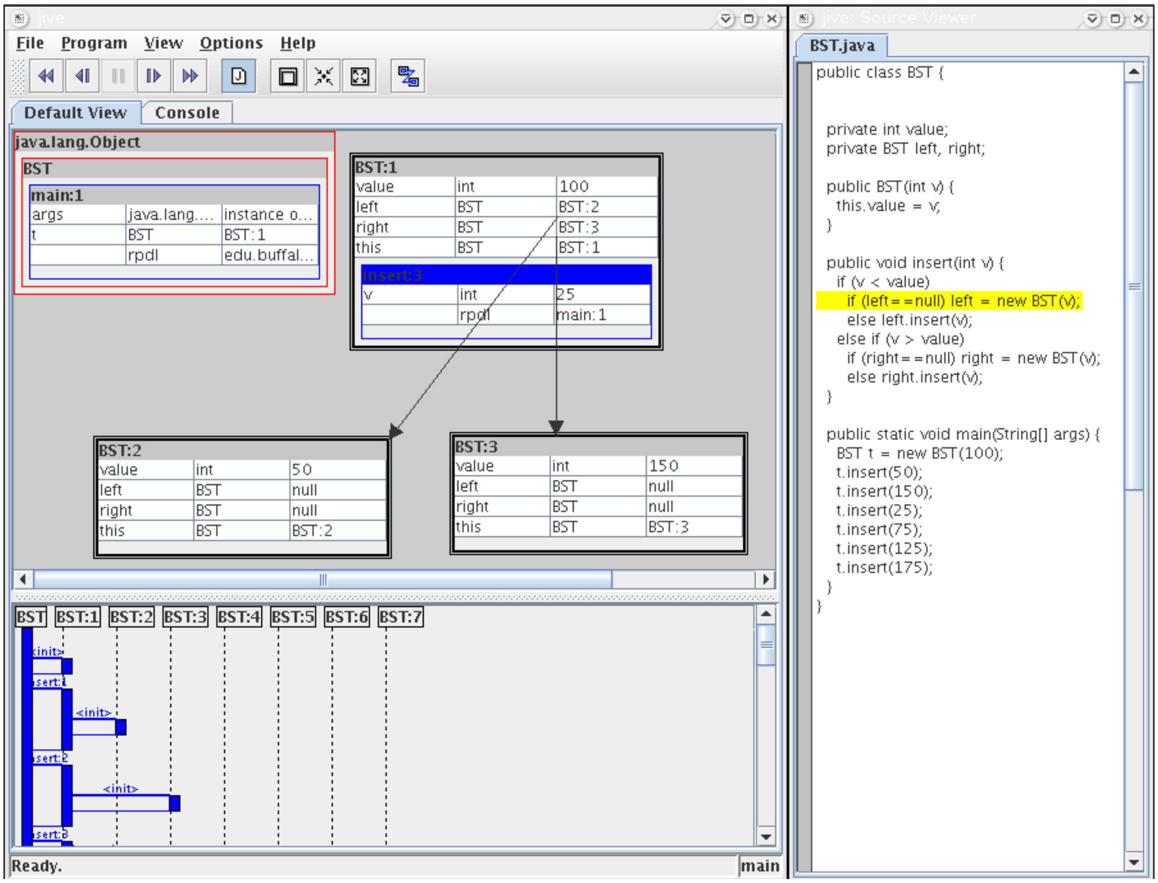


Figure 1.3: This screenshot of JIVE illustrates its major features. In the upper left is a visualization of program state using an enhanced object diagram. In the lower left is a visualization of program history using a sequence diagram. On the right is the source code of the program, and the current line of execution is highlighted. The toolbar at the top of the window is used to control program execution as well as the visualization mode.

highlighting its concurrent visualizations of program execution state, execution history, and the location-highlighted source code of the program.

1.3 Research Contributions

In this dissertation, we introduce a novel visual operational semantics for Java that highlights the fact that objects are environments. This notation supports all major language features, including objects, methods, threads, exceptions, inner objects, and the separation of static and instance space. This notation is used for visualizing a program's state of execution in JIVE. This approach is significantly superior to other formal approaches such as Unified Modeling Language (UML) object diagrams [15], which do not clarify the execution context of methods. Our notation is inspired by the Johnston's classical contour model [55] and extends the work of Jayaraman and Baltus [53].

The visual operational semantics for Java is used in *enhanced object diagrams*, which represent individual Java program states. We provide a thorough analysis of the aesthetic preferences for these diagrams, and we introduce a hierarchical layered drawing algorithm that satisfies our aesthetic constraints and runs in quadratic time. Our algorithm has comparable space and time requirements with existing techniques for layered drawing, but it produces better drawings with respect to our aesthetic constraints.

We apply UML sequence diagrams as a means for visualizing program history. These diagrams are traditionally used for system design, specification, and documentation; by dynamically generating these diagrams, we help close the loop between design and execution. We specify a set of formal aesthetic preferences for sequence diagrams, and we prove that satisfying them all is NP-hard. In order to produce aesthetically good drawings, we introduce three techniques for automatic drawing of sequence diagrams. These are: a very fast first-come, first served approach, which is effective during interactive visualization; a heuristic simulated annealing approach, which is effective for offline visualization; and a hybrid technique that exploits object clusters. The JIVE methodology provides the first visualization approach that combines automatically-generated sequence diagrams with aesthetically and computationally effective drawing strategies.

Both object and sequence diagram drawing algorithms can be enhanced by incorporate an analysis of a program's class diagram. Certain properties in the class diagram can be used to predict the types of object structures that can emerge at runtime, and the class diagram also provides important information about the semantics of these object structures. We introduce a technique that uses class diagram analysis to predict and detect two types of object clusters in the object diagram: leaf clusters and recursive clusters. The cluster prediction and detection take quadratic time with respect to the size of the input, and therefore they do not impact the asymptotic complexity of our drawing algorithms. Once these clusters are detected in the object diagram, each cluster can be drawn using a technique that is specific to that cluster's shape; this is a significant improvement over existing clustered drawing strategies, such as `dot` [33], which use the same technique recursively on each cluster.

The presented architecture for JIVE does not rely on custom virtual machines or compilers. This is done by using a combination of the standard Java libraries with the Java Platform Debugger Architecture (JPDA) and the Apache Byte Code Engineering Library. These technologies are combined within the JIVE architecture, which represents program execution history using a relational database. This provides scalability and the capacity for executing queries on the program state and history using standard query languages such as SQL. The architecture for interactive visualization supports forward and reverse execution using an incremental state-saving paradigm. We show that this is the optimal approach for stepping (in either direction) between arbitrary consecutive pairs of states.

1.4 Dissertation Outline

The previous section gave an overview of the technical approach taken in this work and the JIVE project, and it shows how our approach meets the desiderata for interactive visualization of object-oriented programs. The remainder of this dissertation is organized as follows:

- Chapter 2 describes several projects related to this work. The related work is divided into graph drawing methodologies, program visualization tools, and teaching environments.

- Chapter 3 presents our visual operational semantics for Java. This notation is designed to clarify the fact that, in object-oriented programming, objects are environments of execution. It builds upon the earlier work of Jayaraman and Baltus [53], applying the ideas of their notation to the Java language.

- Chapter 4 demonstrates how the JIVE tool and methodology can be used to enhance program understanding. This is done through four detailed examples taken from students' experiments using JIVE.

- Chapter 5 describes the methodology and software architecture of the JIVE system. Each of JIVE's major components are explored, along with discussions of design decisions, alternate approaches, and efficiency.

- Chapter 6 presents the precise semantics for interactive execution, which can be briefly described as the capacity for forward- and reverse-execution. Implementation notes are included with the discussion of the formal model.

- Chapter 7 describes the problem of automatic drawing of enhanced object diagrams. These diagrams are used to clarify individual states of program execution, and they realize the visual operational semantics for Java presented in Chapter 3. The complexity of the problem is analyzed, and techniques are shown for implementing a graph-theoretic solution.

- Chapter 8 describes the problem of automatic drawing of sequence diagrams. Sequence diagrams depict the history of program execution in terms of the messages sent between objects.

Two solution strategies are described: a stochastic method using simulated annealing and a greedy approach that is very efficient in practice.

- Chapter 9 describes how the graph-theoretic drawing techniques of Chapters 7 and 8 can be combined with program-specific techniques. This approach identifies groups of objects, organizes them into clusters, and then uses clustered drawing techniques to generate diagrams that more closely represent the mental model of a program's execution.

- Chapter 10 presents some conclusions and general observations about the JIVE methodology along with several directions of current and future research.

Chapter 2

Background and Related Work

This dissertation on interactive visualization of object-oriented programs builds upon a broad range of work ranging from language semantics to system architecture to automatic graph drawing. In this chapter, we explore the fundamental background on which the JIVE methodology is built, and we provide the context of this work in relation to the literature and the state of the art. The rest of this chapter is structured as follows;

- Section 2.1 describes *contour model semantics*, an adaptation of which is used in the JIVE methodology to visualize program states.
- Section 2.2 describes sequence diagrams, which are used in the JIVE methodology to visualize execution history.
- Section 2.3 provides a brief introduction to graph drawing research, presenting the fundamentals necessary to understand the automatic drawing techniques we present in Chapters 7 through 9.
- Section 2.4 provides a survey of related work.

2.1 Contour Model Semantics

A programming language can be defined by its *syntax* and *semantics*. The syntax of a language is defined by a grammar, which specifies the set of all well-formed programs for the language. The semantics of a language provides the meaning of syntactically-correct programs. In 1971, John B. Johnston proposed a visual operational semantics for block-structured programs, which he called the *contour model* [55]. He described a program as being composed of *program units*, which can be functions, records, arrays, and so on. Essentially, a program unit can be any lexical unit in which elements

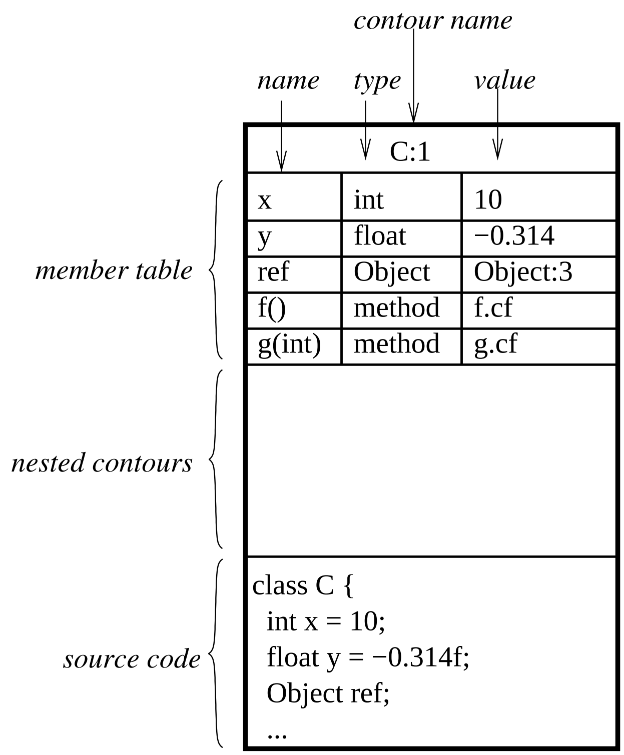


Figure 2.1: The basic syntax for a contour.

share a common scope. Johnston showed how the execution of each program unit could be interpreted as a *contour*, and these contours together form the contour model. The contour itself records the execution-time information relevant to its corresponding program unit, such as the local variables and return point of a procedure. The contour model then is a graphical representation of a program's state, and a *contour diagram* is a drawing of the contour model. Since the contour model provides an abstract machine for expressing the meaning of a program, it is an *operational semantics*. Johnston basically described the operational semantics of procedural languages using contour diagrams. However, the notation and terminology followed in this dissertation is closer to that of Jayaraman and Baltus, who showed how to apply the concept of contours towards clarifying the semantics of object-oriented programming [53].

Basic Syntax for Contour Diagrams

The archetypal structure of a contour is shown in Figure 2.1. Each contour is named and contains three major parts: a member table, an area for nested contour containment, and an instruction list or source-code attachment. The exact structure and content of a contour depend on the language being represented, the program unit to which the contour corresponds, and the state of execution

```
int factorial(int n) {
  if (n==0 || n==1) return 1;
  else return n * factorial(n-1);
}
void main() {
  int result = factorial(3);
  printf("%d\n", result);
}
```

Listing 2.1: Factorial function in C

being shown in the contour diagram; some contour parts may not be included in some contours. For example, a statically-scoped language that disallows methods from being defined within other methods would not have nested contours within its method contours.

The name of a contour must be unique with respect to its enclosing context, whether that be another contour or the global space. It is not necessary that contour names be globally unique since, given the previous constraint, each contour can be uniquely identified by specifying its nesting context. Some presentations of contour diagrams show the name of the contour outside of the rectangular boundary [*e.g.* 52]. This is a minor visual difference and has no other bearing on contour model semantics. In this presentation, we adopt the internal placement of the contour name since this structure simplifies computation of a contour's bounding box and hence facilitates more efficient automatic drawing (see Chapter 7).

The *member table* contains three columns that describe the *members* of the contour. The three columns of the member table are, from left to right, the *name*, *type*, and *value* of each member. This ordering of columns reflects the notation used in the UML. For example, an integer variable `i` with the value 10 would be written in the UML as "i:int=10," which is a name-type-value ordering. The value cell of uninitialized variables, such as unused local variables, is left empty, and unknown values are represented using a question mark ("?") in the value column. The values of non-variable members are expressed as *contour formats*, which are conceptual contour prototypes. These are listed in the value column with the name member and a ".cf" appendix for **c**ontour **f**ormat.

As a concrete example, consider the program of Listing 2.1, an implementation of the factorial function in C. The `factorial` and `main` functions are the two program units that can be translated into contours in the contour model. Figure 2.2 presents a sample contour diagram for the program. Each of the four contours represents an activation of a method, and the name of the method corresponds to the name of the contour. The numbers within the contour names are used to uniquely identify the contour, and these correspond to the order in which the methods were called. The

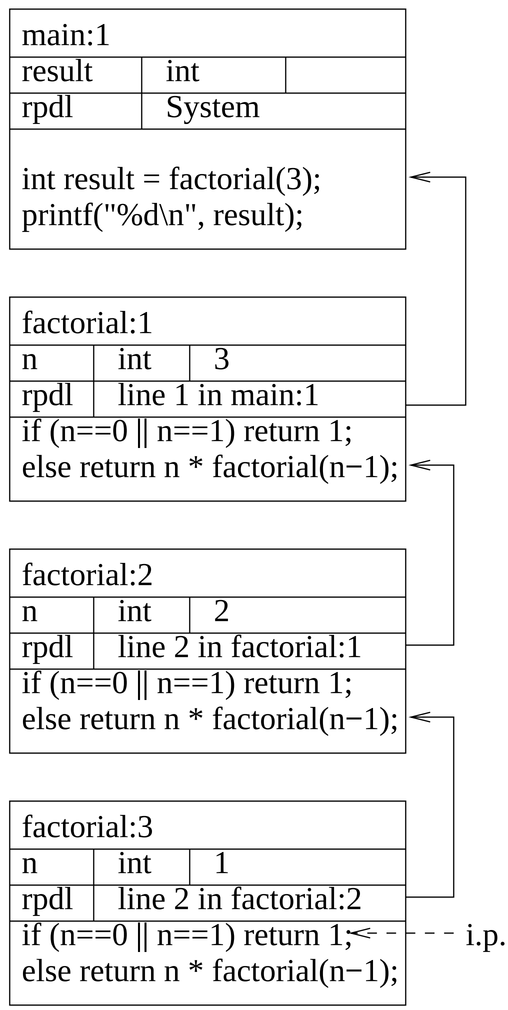


Figure 2.2: Contour diagram for the factorial program of Listing 2.1. This contour diagram represents the state of the program's execution in which `factorial` has been called with the argument 1. The instruction pointer (*i.p.*) indicates the current point of execution. When each of the `factorial` contours returns, the value 6 will percolate through the model and be assigned to `result` in `main:1`.

```
int n = 3;
int factorial(int n) {
  if (n==0 || n==1) return 1;
  else return n * factorial(n-1);
}
void main() {
  int result = factorial(n);
  printf("%d\n",result);
}
```

Listing 2.2: Alternate factorial implementation in C

instruction pointer (*i.p.*) references the current locus of execution; in this case, it is the first line of the third call to `factorial`. Each `factorial` contour clearly indicates the value of its parameter `n`; the value of `result` in `main:1` has not yet been set and so is left blank. Each function contour has a special entry within its member table, the *rpdl*, standing for **r**eturn **p**oint and **d**ynamic **l**ink. The rpdl indicates which contour receives execution focus when the function exits, and the rpdl is rendered here as both text and an arrow.

Identifier Scope

In the classical contour model, there is one rule for resolving the scope of any identifier. Given an identifier i in contour c, first search the member table of c for the i, and if it is found, this is the binding for i. Otherwise, recursively check each containing contour's member table for i; the first that contains it is the correct resolution. As an example, consider Listing 2.2 an alternate form of the factorial program in C; a global variable has been introduced. Figure 2.3 provides a contour diagram for the state of the program where `factorial` has been called for the first time. Notice that a new, unnamed, global contour has been introduced. This contour contains the globally-scoped identifiers `n`, `main`, and `factorial`.

Figure 2.3 illustrates how the contour scope rule is applied. Within `main:1`, there is a reference to an identifier `n`. The definition for `n` is not contained with `main:1`, and so, in accordance with the scope resolution algorithm, we move to the contour containing `main:1`. There is an identifier `n` defined in this contour (the global space contour), and so we know that "`n`" within `main:1` refers to the global `n`. To contrast, consider `factorial:1`, which also references `n`. Here, there is a definition for `n` within the member table of `factorial:1`, and hence the "`n`" in this context is the local variable (formal parameter) `n`.

The scope resolution algorithm has important implications for how contour model semantics are applied to statically scoped and dynamically scoped languages. In statically scoped languages,

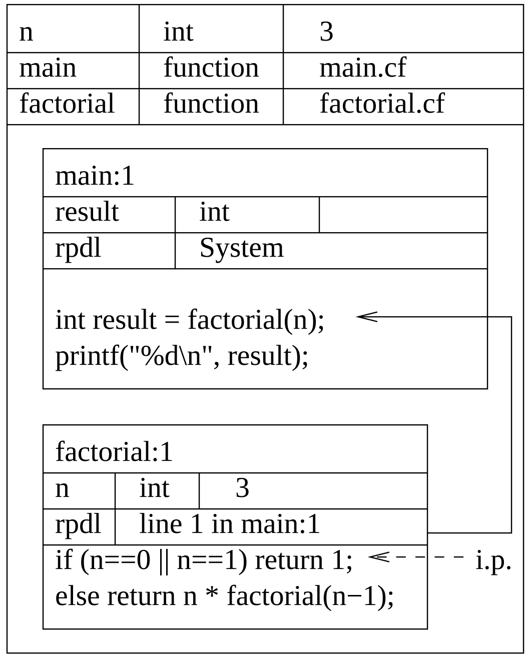


Figure 2.3: Contour diagram for an execution state of Listing 2.2. Compared to Figure 2.2, this contour diagram has introduced an unnamed contour for global space. The member table of this contour contains the definitions of globally-scoped elements.

function activation contours are nested within the context in which the function is defined. Since C is a statically scoped language, Figures 2.2 and 2.3 apply the semantics for static scoping. In dynamically scoped languages, function activation contours are nested within the context in which they are called. Listing 2.3 uses a fictional dialect of C that uses dynamic scoping instead of static. Figure 2.4 illustrates a contour diagram for this program. Here, the value of `debug` in `main:1` trumps the "global" `debug`. This example illustrates the flexibility of contour model semantics, even though the languages we actually address in the JIVE methodology are statically scoped.

```
int debug = 0;
int factorial(int n) {
  if (debug) printf("n=%d\n",n);
  if (n==0 || n==1) return 1;
  else return n * factorial(n-1);
}
void main() {
  int debug = true;
  int result = factorial(3);
  printf("%d\n",result);
}
```

Listing 2.3: Implementation of factorial in a dynamically-scoped, C-style language.

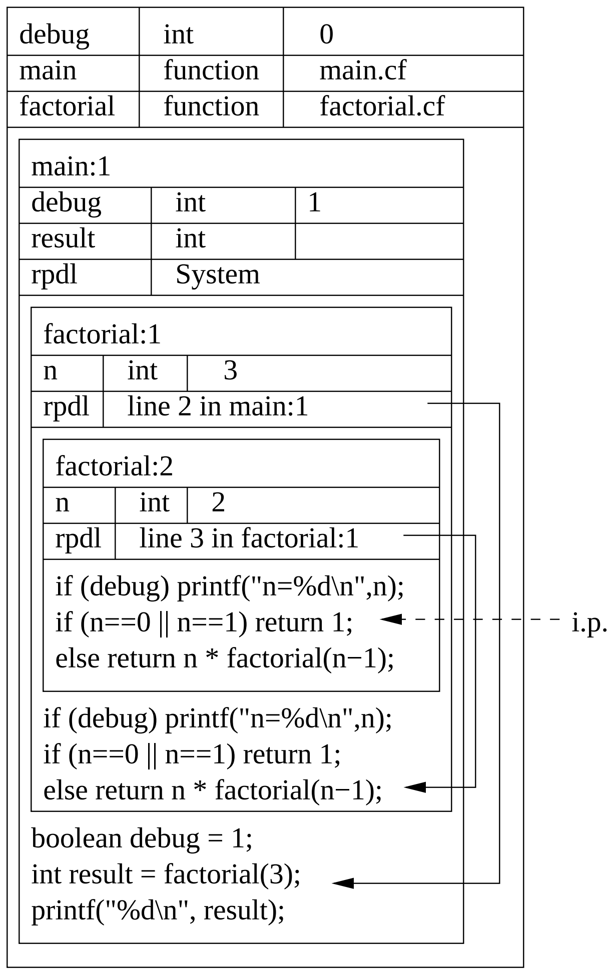


Figure 2.4: Contour diagram for the dynamically-scoped, C-style program of Listing 2.3. This figure illustrates the contour model semantics for dynamic scoping: within `factorial`, the value of `debug` is true since the identifier resolves to the declaration in `main:1`. If this were a statically scoped language, "`debug`" would resolve to the global declaration, and the method contours would be nested flatly within the global space, as in Figure 2.3.

```c
struct cartesian { int x, y; };
typedef struct cartesian POINT;

int equals(POINT* p1, POINT* p2) {
  return (*p1).x==(*p2).x && (*p1).y==(*p2).y;
}

void main() {
  POINT* p = (POINT*)malloc(sizeof(POINT));
  (*p).x = (*p).y = 10;
  int result = equals(p,p);
  printf("%d\n",result);
}
```

Listing 2.4: C program using dynamic allocation

Dynamic Allocation

The previous examples only used statically allocated space for variables. For example, the global variables of Figure 2.3 are allocated at program start, and the parameter of the `factorial` method is allocated on each invocation. Contour model semantics supports dynamically allocated memory in an intuitive manner. When memory is dynamically allocated, a contour is created for the corresponding program unit. Method contours are created in a conceptual *stack space*, whereas contours for dynamically allocated structures are allocated in *heap space*. The two spaces are not physically separated in the contour diagram, since the difference is behavioral, not structural. Specifically, contours allocated in stack space are destroyed when they lose focus (*i.e.* when the function returns), and contours on the heap remain until they are explicitly deleted or the program itself terminates.

Listing 2.4 provides a C program that uses dynamic memory allocation. The `main` method allocates a `POINT` object on the heap, and then passes references to it to the `equals` method. Figure 2.5 is a contour diagram of the program at the point where equality is being tested through the `equals` method. The contour `POINT:1` has been dynamically allocated through the `malloc` of line 1 of `main`. In the contour diagram, both `p1` and `p2` in `equals:1` alias the same object on the heap, making the equality testing somewhat moot in this case. However, the fact that a simple equality by reference would have sufficed is clarified through the contour model semantics.

Interactive Contour Diagram Visualization

The expressiveness and generality of the contour model makes it a useful tool for explaining programming language semantics. Inspired by this, the Language Research Group[1] at the University

[1] `http://www.cse.buffalo.edu/LRG`

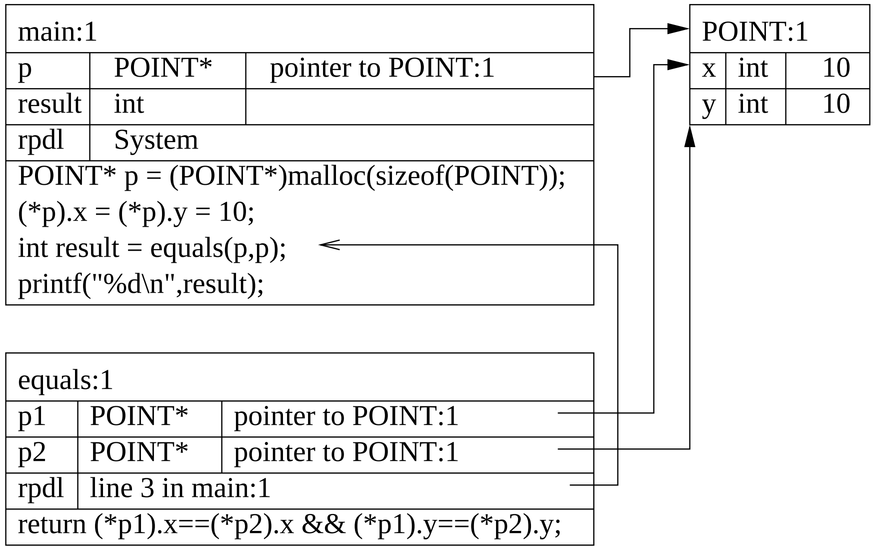


Figure 2.5: Contour diagram for a state in the program of Listing 2.4. The contour `POINT:1` is dynamically allocated and referenced through pointers `p`, `p1`, and `p2`.

at Buffalo developed a contour visualization tool unofficially titled *VISUAL*. The goal of this tool was to provide contour model visualizations in an interactive execution environment. Figures 2.6 and 2.7 provide screenshots of VISUAL, and the JIVE methodology is a direct descendant of this work.

The language modeled in VISUAL is a variation of C with nested and higher-order procedures as well as several different parameter passing paradigms, making it a useful tool for explaining programming and contour model semantics. The grammar for the language includes a special `print` function that, when called, causes VISUAL to open a dialog box with the designated output.

However, the tight coupling of the language with the visualization environment comes at a significant cost: the programs cannot be run outside of the visualization environment at all. This drawback inspires the JIVE methodology's desiderata to support a "real" programming language that is used in software production environments. This requires significant modifications to the contour model, as described in Chapter 3.

Interactive execution in VISUAL involved the ability to step forward or backward through program execution. The program itself was compiled and interpreted from within VISUAL itself. At each discrete step, a copy of the execution state was copied to an ordered list, and so stepping backward became a matter of restoring the previous state in the list. This approach was usable for small programs but it clearly does not scale since the amount of space required is roughly quadratic in the size of the program's contour model and the number of steps in its execution. However, this

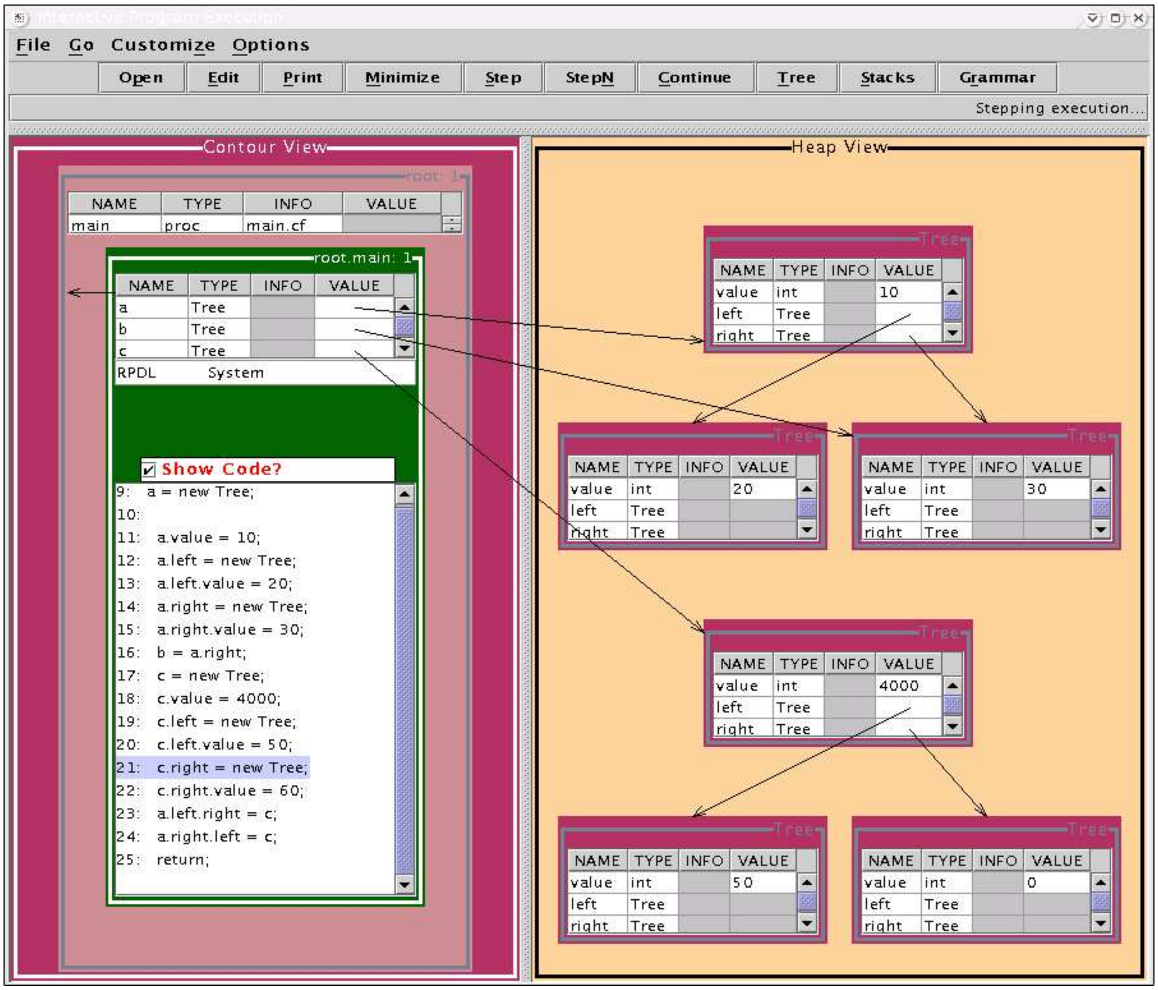


Figure 2.6: A screenshot of *VISUAL*, the original interactive contour visualization tool. VISUAL provides automatic drawing of contour diagrams in the context of interactive forward and reverse execution. The left pane shows stack space, and the right shows heap space. This specific example illustrates insertion into a binary search tree, and the procedure contour on the left shows the current location of execution through source-code highlighting.

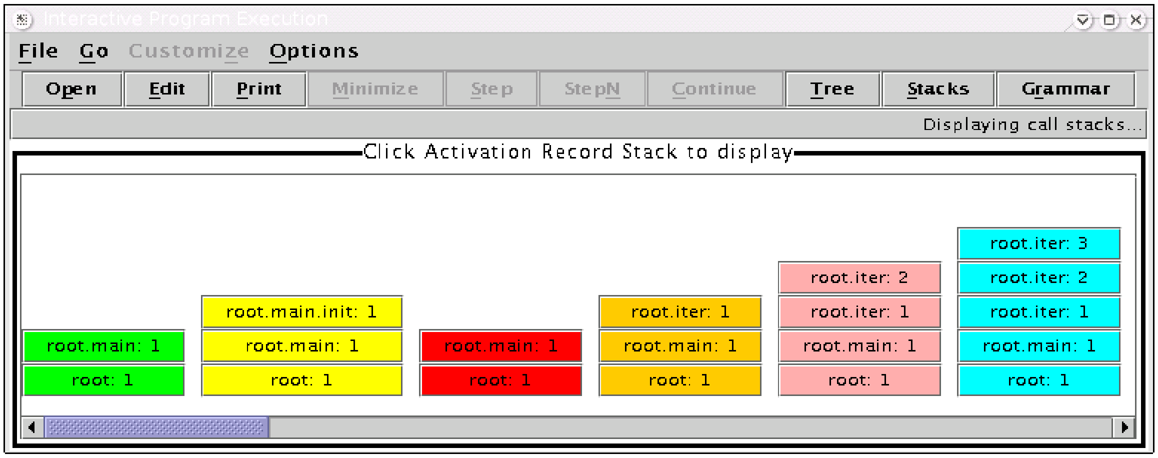


Figure 2.7: Another screenshot of *VISUAL*, this one illustrating *stack view*, which shows the history of frames on the call stack. Selecting any stack will shift the contour view (*e.g.* Figure 2.6) to the appropriate state visualization; in this sense, the call stack is a procedural predecessor to the interactive sequence diagrams of JIVE.

technique does provide for steps between non-adjacent states with the same efficiency as stepping between adjacent ones, which results in a predictable user experience.

2.2 Sequence Diagrams

A *sequence diagram* is a visual tool for describing the a sequence of interactions between entities within a system. The entities are usually objects or users, and sequence diagrams are used in one of two ways, either (a) during system design to specify how entities should interact, or (b) to describe the interactions of systems in distributed documentation. These diagrams have proven to be an effective method for expressing dynamic system behavior, as described in the work of Hahn and Kim [44].

We will use the Unified Modeling Language specification for sequence diagram syntax as described in Booch et al. [15]. In a UML sequence diagram, time increases downward along the vertical axis, and entities are listed along the horizontal axis. An example is shown in Figure 2.8. There are three basic components of the sequence diagram syntax:

Lifeline The lifeline is a vertical dashed line that extends from an entity's creation to its destruction. The name of the entity represented by the lifeline is marked at its top. This entity is most often either an object or an actor.

Focus of control The focus of control represents the time at which a specific entity is performing an action. It is represented as a rectangle over the corresponding entity's lifeline that stretches

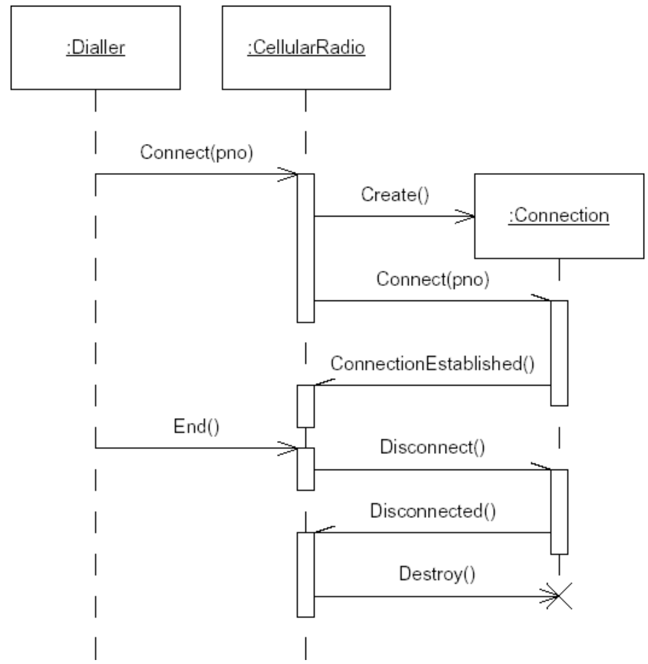


Figure 2.8: Sample UML sequence diagram. This diagram illustrates the communications among a dialer, cellular radio, and connection object. Synchronous messages are shown with full arrowheads and asynchronous messages are shown with half arrowheads. This example is taken from Robert C. Martin's sequence diagram tutorial [68].

over the time period of the action.

Message A message represents communication between entities, and it is depicted in the sequence diagram as a directed arrow between two lifelines.

Like other UML diagrams such as class and use-case diagrams, sequence diagrams benefit from a strong capacity for abstraction. A lifeline may represent a module, in which case the focus of control represents the time the module is processing; likewise, a lifeline may represent an object, and the focus of control can represent a method activation on that object. There is also a rich syntax and semantics for messages. Figure 2.8 illustrates three types of messages. The full arrows represent *synchronous messages*, that is those messages that block until a response is received. The half arrows represent *asynchronous messages*, those that deliver the message and return immediately, leaving the message to be processed by the receiver according to its design. Message semantics can be further specified with other types of arrows or by applying stereotypes; for a complete reference to the UML notation for sequence diagrams, see Booch et al. [15].

Sequence diagrams clarify the messages passed among objects as well as the interaction of objects over time. However, there are some disadvantages to using sequence diagrams. Sequence diagrams potentially use quite a bit of space; they quickly become sparse if there are many entities and few

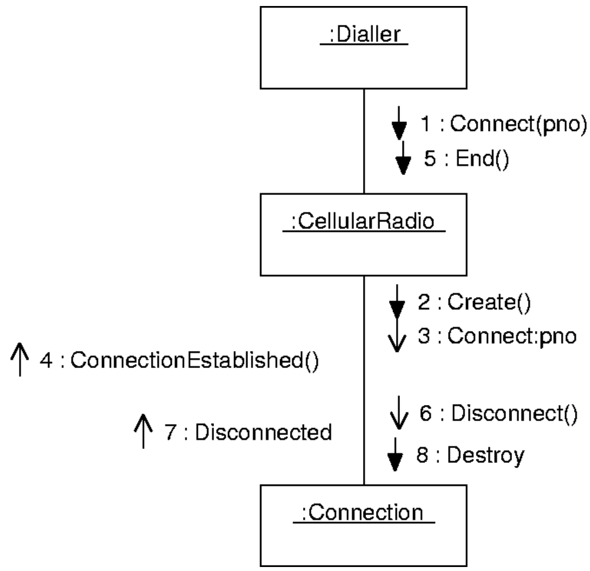


Figure 2.9: A UML collaboration diagram for the same message sequence as in Figure 2.8. The collaboration diagram depicts object linkage, but it does not clarify the order of messages nor the duration of focus of control as well as sequence diagrams. In this figure, filled arrowheads represent synchronous messages and line arrowheads represent asynchronous.

interactions. They also do not show the structural connections among objects. The fact that a message is passed from an object of type `A` to an object of type `B` implies that there is an association between `A` and `B`, but no conclusions can be made regarding aggregation or composition [42]. Other UML diagrams can be used to overcome some of these disadvantages. *Collaboration diagrams*, for example, clarify the connectivity of objects, although they do not indicate the behavior of messages over time as clearly as sequence diagrams; figure 2.9 provides a collaboration diagram to illustrate the same communication protocol as the sequence diagram of Figure 2.8. The UML categorizes sequence diagrams and collaboration diagrams together as *interaction diagrams* due to their similar function in behavioral modeling. In fact, the two contain the same information except for object linkage, which is only specified in collaboration diagrams, and the duration of focus of control, which is only specified in sequence diagrams.

2.3 Graph Drawing

Graph drawing research involves the problem of computationally generating good visualizations of mathematical graphs and similar structures. A drawing is "good" if it meets the desired aesthetic properties and it runs without excessive overhead in time and space. Formally then, graph drawing involves the specification of aesthetic constraints and the development of efficient algorithms for

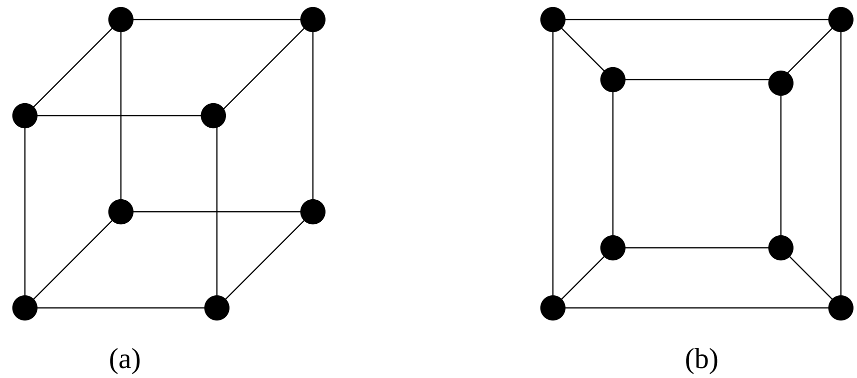


Figure 2.10: Two drawings of the cube graph. This example is taken from Di Battista et al. [24].

solving them. Both the constraint specification and the algorithm development are intrinsically tied to the nature of the graph being drawn. For example, trees have different properties than directed acyclic graphs, which have different properties than general directed graphs. We briefly explore these categories of graphs by way of introduction. For an excellent overview of graph drawing techniques and research, please refer to Di Battista et al. [24].

As an example, consider Figure 2.10, which shows two drawings of the cube graph. The two drawings represent the same graph in two different ways. Figure 2.10(a) contains many crossings and is asymmetric, but it reveals the underlying structure causing this graph to be called the *cube* graph. Figure 2.10(b) contains no crossings and has several symmetries (horizontal, vertical, and two diagonal). Determining which drawing is "better" is a matter of assigning weights to the aesthetics and constraints of the drawing. For example, if the criterion to minimize crossings was more important than the constraint of revealing the cubic structure, Figure 2.10(b) would be chosen as better drawing.

Aesthetic Criteria

The specification of aesthetic criteria involves a combination of domain-specific properties and an understanding of a human's ability to comprehend visualizations. Studies such as those of Batini et al. [11] and Purchase et al. [74] provide important data regarding the comprehensibility of diagrams with respect to aesthetics. Some common graph-drawing aesthetics are:

- *Minimization of edge crossings.* There should be as few edge crossings as possible, as this helps reduce the visual complexity of the drawing. A *planar* drawing, which is one with no crossings, is especially desirable.

- *Minimization of area.* The overall area of the drawing should be minimized. This aesthetic

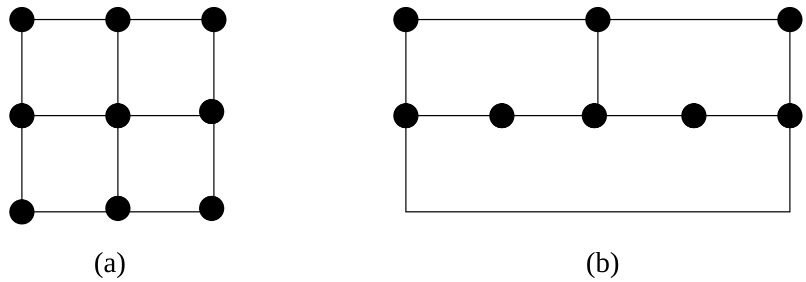


Figure 2.11: Two orthogonal grid drawings of the same graph. Part (a) minimizes the number of bends, and part (b) minimizes the number of crossings. (This example is taken from Di Battista et al. [24].)

only makes sense in the presence of a *resolution rule* such as a minimum unit distance between vertices.

- *Minimization of total edge length.* The sum of the lengths of all edges should be minimized. This reduces the amount of space that must be scanned in order to see and understand the connections between vertices. Like minimization of area, this aesthetic only makes sense under a resolution rule.

- *Minimization of the number of bends.* A *bend* is a non-vertex meeting of two consecutive line segments; bends are introduced when edges are represented as a polygonal chain of line segments. Bends reduce the legibility of a graph, and hence reduction of the number of bends is desirable. Especially desirable is a *straight-line* drawing, which is a drawing with zero bends.

Any nontrivial drawing will use several aesthetic constraints, but these constraints often conflict. For example, consider the two drawings in Figure 2.11. Both are orthogonal grid drawings; that is, the edges are orthogonal, and vertices, edges, and bends have integer coordinates. Figure 2.11(a) has the minimum number of bends, but Figure 2.11(b) has the minimum number of crossings. As with Figure 2.10, the choice of which drawing to use depends entirely on the relative weights assigned to the conflicting aesthetic constraints. Unfortunately, satisfying even proper subsets of these conflicting constraints can be NP-hard. For example, even for a graph as simple as a binary tree, minimizing edge lengths and edge crossings is NP-hard [35], and Garg and Tamassia [36] show that minimizing crossings while maintaining upwardness is NP-hard. Effective heuristics are therefore required in order to construct good drawings within reasonable time; this is especially true for systems such as JIVE that support interactive visualization.

The aesthetic constraints that are desirable for a drawing depend on the nature of the data being represented. For example, Gutwenger et al. [43] have shown that a topology-shape-metrics approach to orthogonal drawing [12] can be used to automatically generate effective drawings of UML class diagrams. We expand on this idea by exploring three well-known and common types of graphs: trees, directed acyclic graphs, and general directed graphs. The discussion assumes that these graphs are directed, but the techniques apply as well to undirected graphs; an undirected graph can always be converted into a directed graph by either assigning a direction to each edge or introducing pairs of opposing directed edges for each undirected one.

Trees

Trees often represent hierarchies, and so it is desirable that a drawing of a tree convey the structure of the hierarchy, thereby facilitating a user's understanding. In addition to the general aesthetic criteria presented above, the following criteria should also be followed when drawing a tree:

- *Upwardness.* Given two nodes u and v such that there is a directed edge from u to v, u should be drawn above v. This ensures that the hierarchical decomposition flows vertically in the drawing, and all edges are aligned in roughly the same direction.

- *Layeredness.* The children of the same parent should be placed on the same y-coordinate; that is, the children are on the same horizontal *layer*. Combined with the previous property, this implies that the layer containing the children will be below the layer containing the parent.

- *Centeredness.* The parent should be centered above its children. That is, the x-coordinate of the parent should be the average of the x-coordinates of its children.

Supowit and Reingold [86] have shown a linear program can be constructed whose solution can give an upward layered centered planar drawing of a tree with minimum area. Unfortunately, the linear program may take polynomial time to solve, and so faster heuristics may be preferable. Reingold and Tilford [76] have developed a widely-used heuristic that constructs an upward, layered, centered, planar, straight-line drawing with small area. Additionally, the heuristic runs in $O(|V|)$ time, where V is the set of vertices in the tree. The basic steps involved in the Reingold-Tilford algorithm are:

1. Recursively construct the drawings the subtrees rooted at each child of v.

2. Place these drawings as close as possible to each other such that their roots have the same y-coordinate, the drawings do not overlap, and the minimum distance between any two nodes is two units.

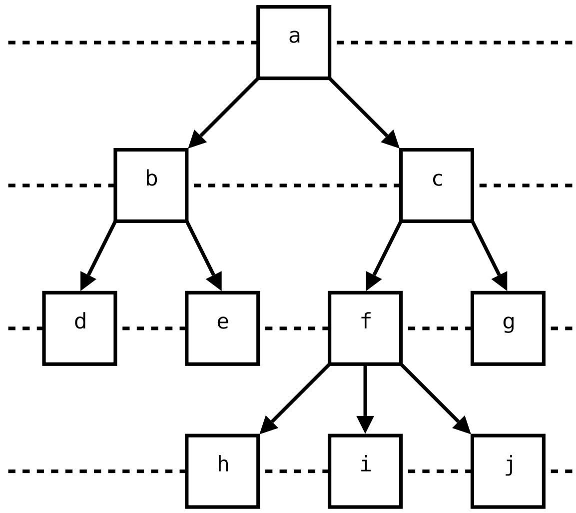


Figure 2.12: Layered tree drawing using the algorithm described by Reingold and Tilford [76]. The horizontal dashed lines represent the layers.

3. Place v such that its x-coordinate is the average of its children's x-coordinates and its y-coordinate is one unit above its children's.

Figure 2.12 provides an example of a diagram generated using the Reingold-Tilford algorithm.

It should be noted that upwardness is not restricted to vertically-aligned drawings where all edges point downward. The upwardness property can hold regardless of the actual alignment used; for example, a common method of drawing single-elimination tournament brackets is a horizontally-aligned drawing where edges point rightward. Such a drawing can be said to exhibit upwardness nonetheless.

Directed Acyclic Graphs

A *directed acyclic graph* (DAG) is a directed graph with no cycles. A DAG defines a dominance relationship between connected vertices, a predecessor/successor relationship that should be captured in its drawing. That is, the drawing should clarify the relationships among vertices. Hence, in addition to the aesthetics presented above, the drawing of a DAG should satisfy the following criterion:

- *Upwardness.* Since a directed edge (u, v) represents a predecessor/successor relationship, u should be placed above v in the drawing.

Although a DAG is combinatorially simpler than a general directed graph, constructing a drawing that satisfies these aesthetic criteria is still difficult. Even testing whether a DAG admits an upward planar drawing is NP-hard, as shown by Garg and Tamassia [36].

A well-known technique for drawing DAGs is the *layered* or *hierarchical* drawing [85]. This technique divides the drawing space into a set of horizontal layers such that each layer is separated by a unit distance. An outline of this approach is:

1. Assign nodes to layers. This assignment is done in accordance with drawing constraints such as upwardness or proximity of connected vertices (that is, for a pair of vertices u and v such that there is an edge from u to v, u and v should be placed as closely as possible).

2. Assign relative horizontal order to the vertices such that crossings between layers is minimized. This step is often solved by considering each pair of adjacent layers and solving the smaller problem of minimizing crossings between those layers. Unfortunately, even this restricted problem is NP-complete [35].

3. Assign x-coordinates to the nodes of each layer, maintaining their relative ordering from the previous step. This is also done according to specific drawing constraints such as minimization of edge length or centering parent nodes above their children.

Gansner et al. [33] have shown that this technique can be implemented using a linear program, which can be quickly solved by the network-simplex method. Figure 2.13 is a layered drawing of a DAG that has used Coffman-Graham layering [21, 24]. The algorithm for drawing trees is in a sense a specialization of general DAG drawing. This is logical since a tree is a specialization of a DAG; specifically, a tree is a DAG with no back edges. Hence, any DAG drawing technique can be applied to trees, although the converse does not hold.

General Directed Graphs

Drawing general directed graphs is more difficult than drawing DAGs or trees. A common approach is to drawing a general directed graph G is to use the following three-step process:

1. Construct a DAG G' from the general directed graph G by removing cycles from G. This is done by reversing a subset of the edges of G.

2. Apply established drawing techniques such as layered drawing in order to construct a drawing of G'.

3. Reverse the transformation of step 1 by reversing the edges of G' to match G.

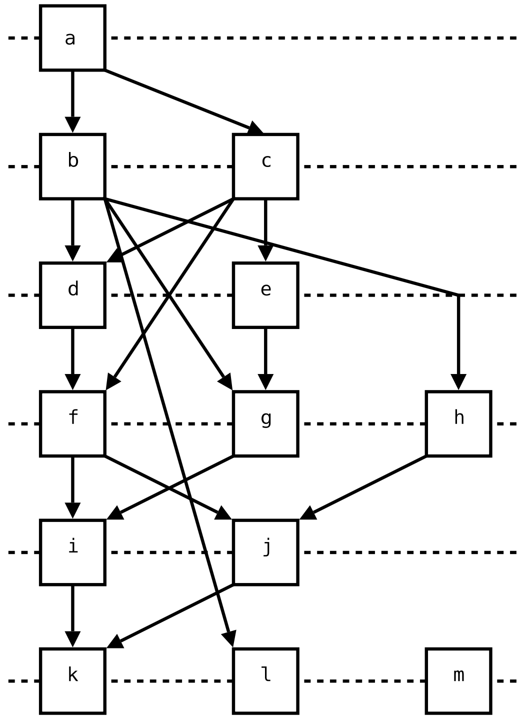


Figure 2.13: Layered drawing of a DAG. This layering was done using the algorithm of Coffman and Graham [21], an algorithm for multiprocessor scheduling. This drawing also illustrates a bend on the arc (b, h). (This figure is adapted from Di Battista et al. [24].)

Note that this technique will not necessarily satisfy the edge-related aesthetic constraints of the algorithm chosen for step 2. For example, converting G' back into G may invalidate upwardness. Furthermore, the problem of choosing the minimum number of edges to reverse in order to convert a directed graph into a DAG is NP-hard, and it is an instance of the *minimum feedback arc set* problem [35]. There are heuristics that can be used to choose good (though not optimal) subsets of edges to reverse. For example, Eades et al. [29] describe a technique that finds a good solution to the minimum feedback arc set problem in linear time with respect to the number of edges.

Nesting Trees and HV-inclusion Diagrams

The techniques described above are for drawing traditional *graphs* that are defined by a set of vertices and a set of edges. However, the contour model semantics for understanding program states is not a traditional graph; instead, it is composed of sets of nested rectangles and the edges between them. The recursive nesting of contours defines a *nesting tree* T_o whose root is o, whose nodes are the contours nested (at any level) within o, and where a contour u_1 is the parent of contour u_2 if and only if u_2 is nested within u_1, and there is no other object or method contour u_3 such that u_3 is

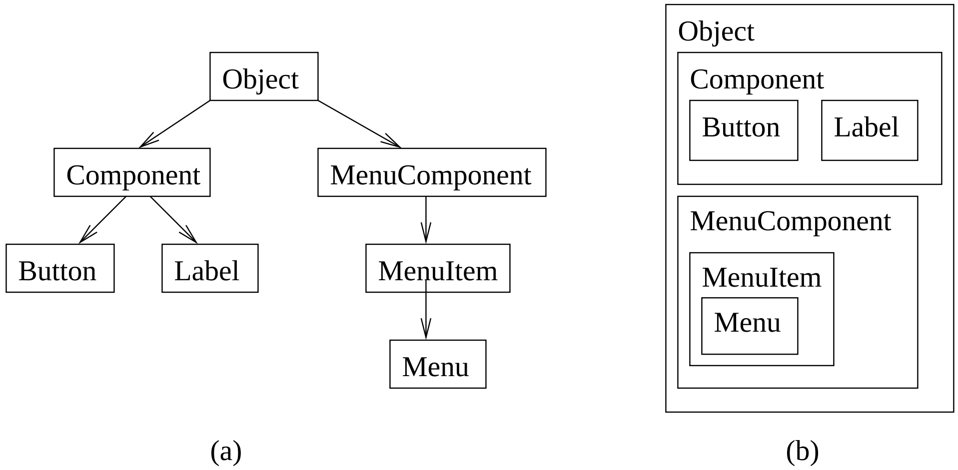


Figure 2.14: A nesting tree (a) and an HV-inclusion drawing of it (b).

nested in u_1 and $u2$ is nested inside u_3. This is illustrated in the nesting tree of Figure 2.14(a) and its drawing as an *inclusion* drawing in Figure 2.14(b). It should be clear from this drawing that there is a strong connection between inclusion drawings and the contour model. In the figure, the nesting tree represents a portion of Java's AWT API, and the inclusion tree corresponds exactly to a contour drawing of static space [37].

The primary aesthetic goal for constructing an inclusion drawing of T_o is to minimize the drawing's area; that is, the area of the root node o should be minimized. Unfortunately, this problem is NP-hard, as described by Lin [63]. To simplify the problem, we restrict ourselves to a special kind of inclusion drawing, an *HV-inclusion* drawing. This is a drawing where, for any node, the drawings of its children are either arranged horizontally (*H-arrangement*) or vertically (*V-arrangement*). The drawing in Figure 2.14(b) is an HV-inclusion drawing.

Although constructing an HV-inclusion drawing with minimum area is NP-hard [63], there are advantages to the HV-restriction. Eades et al. [28] present a dynamic programming based technique for producing an area-minimized HV-inclusion drawing of a binary nesting tree in pseudo-polynomial time. A *binary nesting tree* is a nesting tree in which each node has an outdegree of at most two. This approach takes $O(Mn)$ time where

- n is the number of nodes in T_o, and
- $M = max(\sum u \in T_o w_{min}(u), \sum_{u \in T_o} h_{min}(u))$

where $w_{min}(u)$ and $h_{min}(u)$ are the minimum width and height, respectively, of a rectangle that can display the textual and graphical information associated with a node u that is not contained in its children. The two minima could reflect the maximum dimension of a contour's name, for example.

This approach assumes that $w_{min}(u)$ and $h_{min}(u)$ are provided as input.

2.4 Related Work

JIVE integrates program visualization, interactive execution, and advanced graph drawing into a methodology that is useful as a debugger and as a teaching tool. Each of these domains is rich with examples that address, to varying extents, some of the desiderata for interactive program visualization presented in Chapter 1. In this section, we survey some of the prominent examples and compare their approach to the JIVE methodology. We broadly categorize the related work into the groups given below; this is a generalization that is not meant to be exhaustive or definitive, but merely to provide convenient comparison between related projects.

- *Integrated Development Environments and Debuggers.* Integrated development environments (IDEs) are tools that generally incorporate text editors, compilers, and debuggers into a single platform. Some IDEs are designed for professional development, containing tools for code analysis, refactoring, and round-trip engineering, while others are simplified environments meant to teach the basics of programming and computer science.

- *Program Visualization.* Information visualization is a broad and growing research area, and there are many projects related to visualization of programs. These range from object-level visualizations to heap- and memory-analysis tools to visualizations of software deployment.

- *Interactive Execution.* Several research groups have examined the problem of interactive execution. Different paradigms have been explored, including reverse execution (*i.e.* reversible computations) and state-saving techniques.

- *Graph Drawing.* Graph drawing was introduced in Section 2.3. There are many software tools that implement graph drawing algorithms and allow for both interactive and automatic layout of different types of graphs.

Integrated Development Environments and Debuggers

Integrated development environments provide a single platform through which developers can edit, compile, test, and debug programs. As described in Section 1.1, the debugging facilities of most modern IDEs are still essentially based on text-based debugging techniques. The fundamental problems with traditional debuggers, with respect to our requirements for an effective visual debugger, are that:

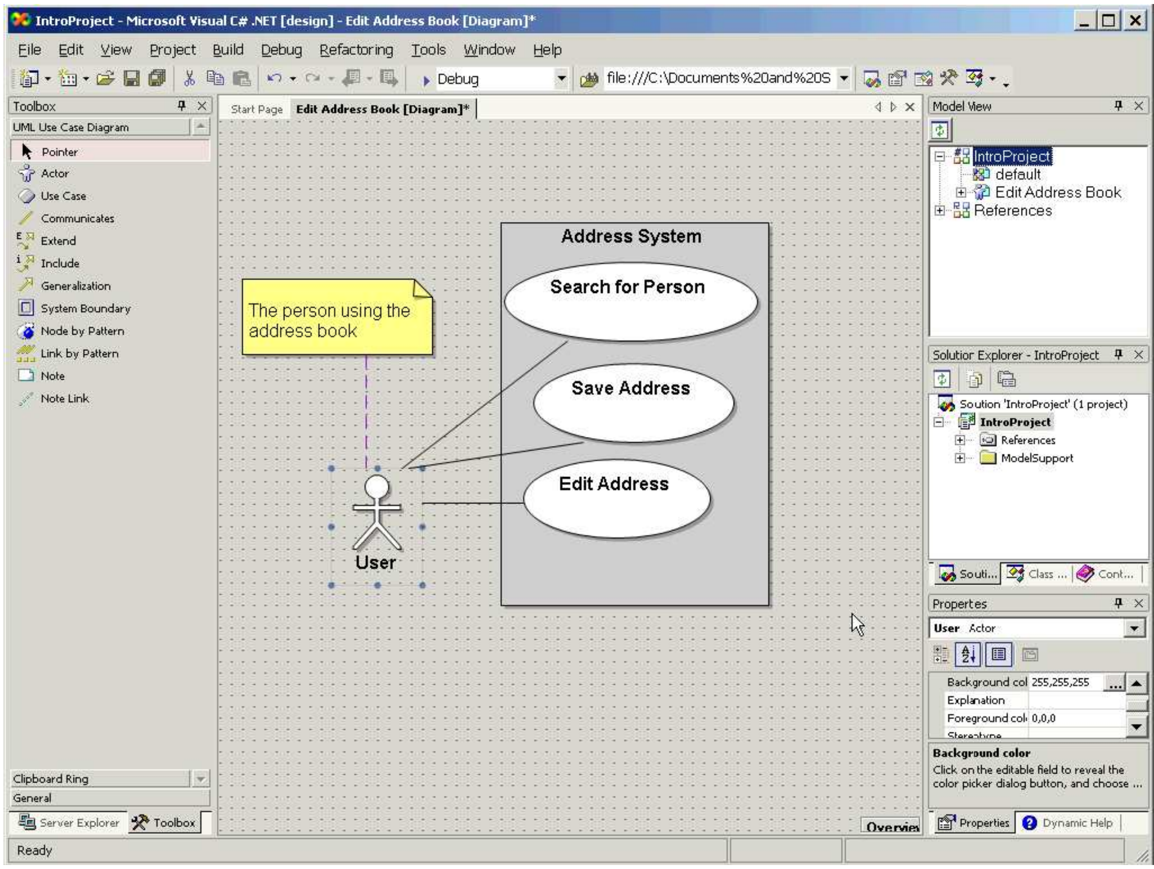


Figure 2.15: Borland Together integrated with Microsoft VisualStudio .NET. This screenshot shows Together's capability for drawing UML use-case diagrams within the IDE.

1. there is no depiction of objects as environments of program execution, or this relationship is obscured through separate visualizations of the call stack and of object contents;
2. there is no clear depiction of execution history (*e.g.* through a sequence diagram);
3. there is no visualization of object structure, or this visualization is essentially flat or tree-based, as in Figure 1.1); and
4. there is no support for reverse execution, meaning that the user can step forwards through execution but not backwards.

The "visual programming" components of modern IDEs are usually nothing more than graphical tools for the construction of graphical user-interfaces (GUIs). Some IDEs, such as IBM's Visual Age tools, provide more semantic-oriented visual tools such as the capability to graphically link publishers and subscribers.Tools and extensions like Rational Rose and Borland Together provide visual tools for program design specification; the latter is shown in Figure 2.15. These provide design tools such as UML diagrams as well as engineering tools that integrate the diagrams into the overall software development process. Some tools use an *ad hoc* integration while others follow formal models such as the Rational Unified Process [62]. While these types of applications do ease design, development, and testing, they do not provide equally advanced tools for visualizing program execution. Design tools such as the UML or the Rational Unified Process focus on specification and design, but they do not address debugging and runtime understanding with the same weight.

The Smalltalk language is designed to be integrated into an interactive IDE, and it pioneered many of the patterns of modern IDEs and graphical user interfaces in general. Smalltalk's unique design incorporates a fully-observable and modifiable virtual machine [58]. The Smalltalk Inspector provides a view of objects similar to the visual semantics we propose: it provides a usable visual interface through which the details of objects and their inheritance hierarchies can be explored [see *e.g.* 40]. Figure 2.16 provides a screenshot of Squeak's object inspector. Recent projects such as Squeak[2] and Croquet[3] show how the Smalltalk language and environment has evolved into modern multimedia and network-aware, immersive environments [50, 83]. However, Smalltalk IDEs generally do not provide a view of program history, nor do they visualize object graphs or methods in their inherited object contexts. We believe that the concept of an interactive Inspector is fundamentally important, and this is a strong influence on the JIVE methodology for visualizing object-oriented program states.

Pedagogic IDEs are designed to help students learn the basics of programming, computer science, and software engineering, and hence they tend to have a very different interface than professional IDEs. Perhaps the most well-known pedagogic integrated development environment is BlueJ[4], currently maintained by a joint research group at Deakin University (Melbourne, Australia) and the University of Kent (Canterbury, United Kingdom). BlueJ is an integrated development environment designed for use by students of computer science in their introductory-level courses. It is designed to be used with an objects-first pedagogy, as described in the recent book by Barnes and Kölling [8]. BlueJ features visualizations of program architecture through class diagrams, graphical and textual editing of source code, and interactive object creation and method invocation, which provides an environment incremental application development. Recent studies have proven the effectiveness of BlueJ in the classroom [88].

Figure 2.17 presents a screenshot of BlueJ. The application being executed displays bouncing balls on a canvas (not pictured); it is a demonstration by Bruce Quig that was taken from the

[2]http://www.squeak.org
[3]http://www.opencroquet.org
[4]http://www.bluej.org

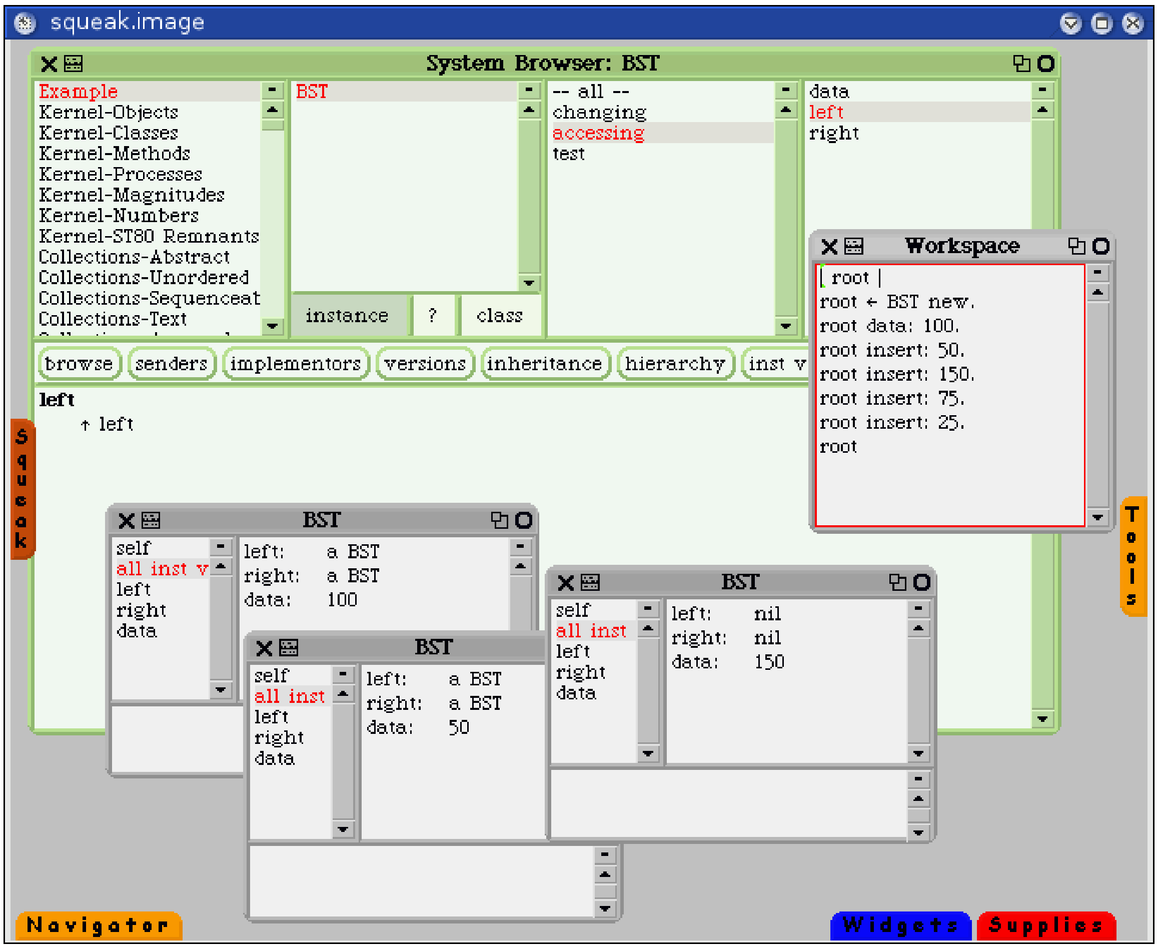


Figure 2.16: A screenshot of Squeak. The three similar windows in the foreground are inspectors, and they are being used here to show the details of a binary search tree, specifically the root and its two children.

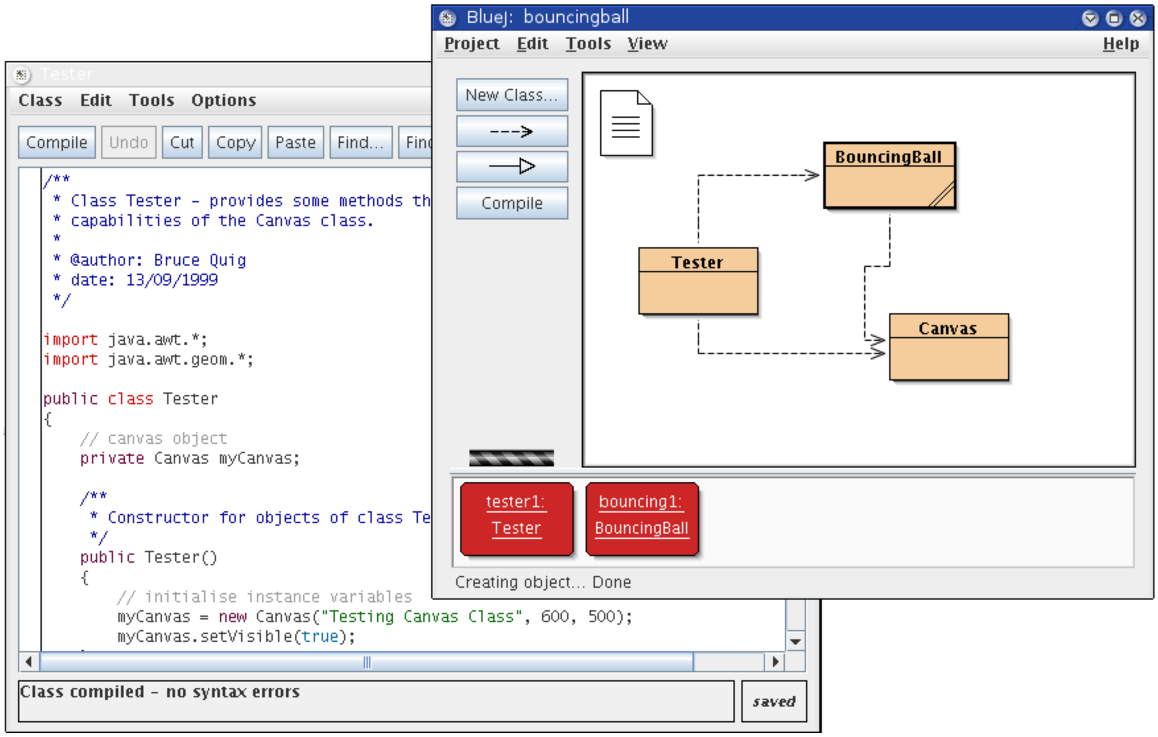


Figure 2.17: A screenshot of BlueJ. In the background is the source code of the program. The foreground window presents a UML-style class diagram representing the program structure, and at the bottom are the currently allocated objects.

BlueJ web site. The screenshot shows BlueJ's class diagram visualization, which uses a notation similar to the UML. The red boxes along the bottom of the main window show object instances. While the interactive capabilities of BlueJ provide a proven environment for teaching an objects-first course, it does not match many of the criteria described previously. There is no visualization of object structure, and method activations are not shown in their object contexts. The "interactive execution" of BlueJ deals with interactive calls and creations, but not the forward- and reverse-execution we describe.

jGRASP is another development environment designed for educational use [48]; a screenshot is provided as Figure 2.18. It has integrated a visual debugger, and it provides dynamic state visualization. The design of jGRASP facilitates its use as a debugger in introductory computer science courses. Its visualizations are primarily for the integration of graphical design (*a la* UML) for code generation. Also, jGRASP does not include support for interactive execution in the reverse direction, and hence cannot provide for comparative analysis of program histories or querying facilities.

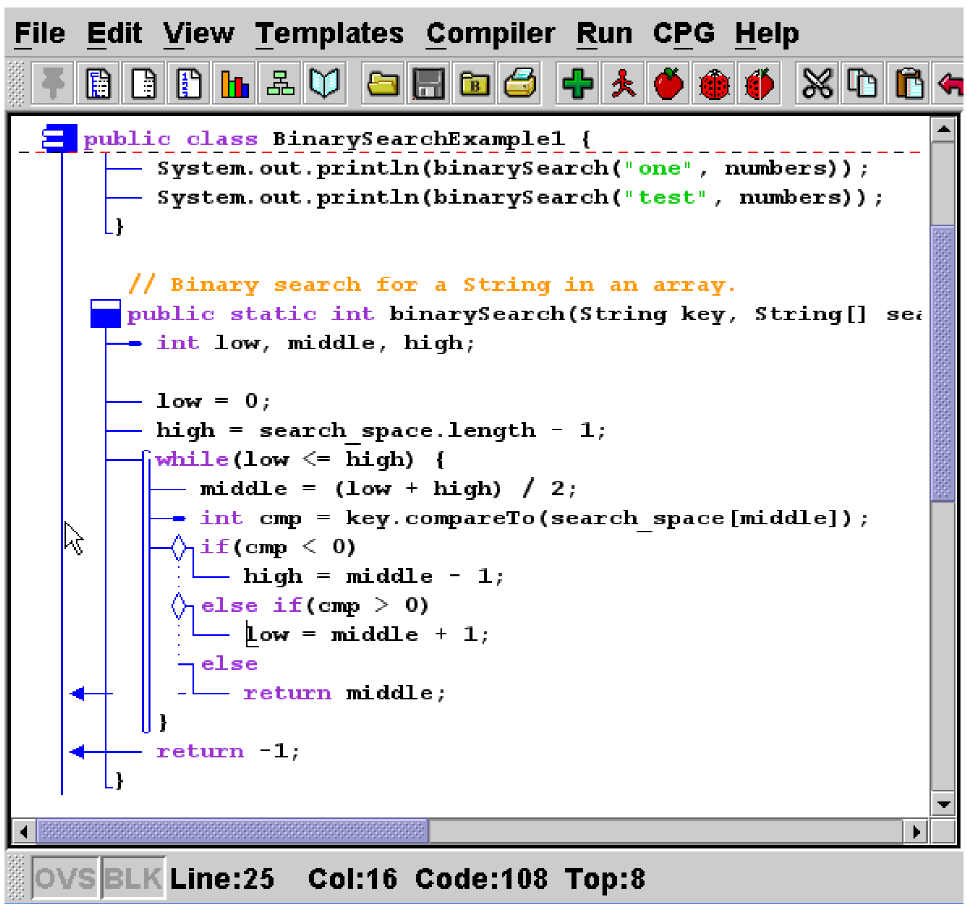


Figure 2.18: Screen shot of jGRASP's Control Structure Diagram (CSD). As this screenshot illustrates, the visualizations of jGRASP are directed towards source-level understanding as part of a pedagogic IDE. This image is adapted from the jGRASP homepage at `www.jgrasp.org`.

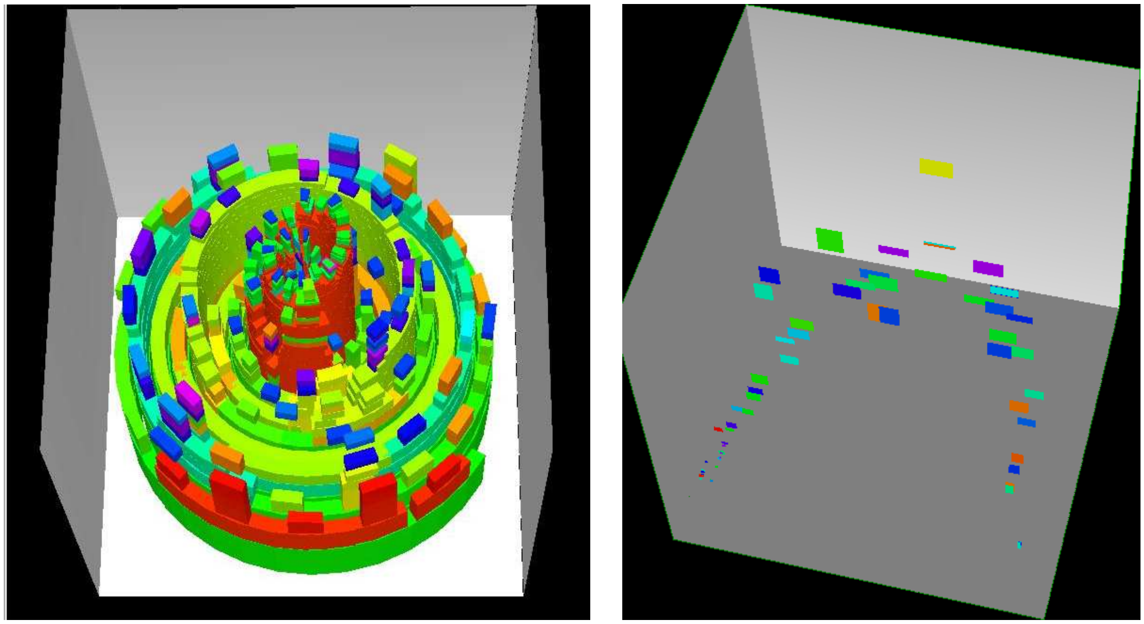


Figure 2.19: Two screenshots of BLOOM. The left image illustrates a spiral view of method call stacks, and the right image displays allocations by class. Both images are taken from the BLOOM web site at `http://www.cs.brown.edu/people/spr/research/vizbloom.html`.

Program Visualization

Information visualization generally refers to any conversion of raw data into a visual form. Examples range from visualizations of wiki activity to database transactions. Program visualization, in this context, refers to the application of information visualization techniques to the domain of programming. The complexity of modern software makes such visualizations desirable since they facilitate design, debugging, and communication.

BLOOM is a robust visualization tool that includes a variety of advanced trace collection and analysis techniques [77], and it has been developed at Brown University by Steven Reiss. BLOOM uses three-dimensional visualizations in order to show such execution metrics as object allocations over time, call graphs, and performance data. Figure 2.19 provides two screenshots of BLOOM visualizations. BLOOM is explicitly focused on high level runtime visualizations, and therefore produces visualizations very different from the ones we describe. Despite this variance in focus, BLOOM research has also predicted the usefulness of visual query languages, and such facilities have been incorporated into the tool [78]. JIVE[5] is another project under Reiss' supervision, and it is similar to BLOOM. JIVE provides low-overhead, high-content visualizations of execution information, and it is targeted specifically for Java programs [79]. Matrix Visual Tester (MVT) is similar to JIVE

[5] To avoid confusion, we typeset Reiss' JIVE in full capital letters and our homonymous tool in small capitals.

in its design and intent [66]. It is designed as a visual debugging aid, and it supports forward and reverse execution. A screenshot of MVT is shown in Figure 2.20; this screenshot illustrates the visual paradigm of MVT's state diagrams (on the left) and package hierarchy view (on the right). Both MVT and JIVE rely on abstraction and elision, automatic and user-controlled, to reduce the amount of information presented in the visualization. MVT uses bytecode instrumentation and allows the user to dynamically change variables, whereas JIVE uses unmodified class files and does not allow for dynamic changes to programs. To contrast, JIVE does not perform any re-execution, and therefore avoids the viscosity problem, which occurs when a user steps backward and makes a change that invalidates recorded states; MVT also does not support interactive queries through an execution database.

The *VisBug++* environment was designed as a visualization tool for C++ and to show the benefits of using visualization with object-oriented languages [54]. Figure 2.21 provides a screenshot of VisBug++. This system is restricted to the visualization of method activations as program units. Our work builds upon this by providing complete visualizations with statement-level granularity. Many of the ideas put forth in this previous research have been applicable to our own work, including the benefit of "rewinding" execution states and the application of four central visualization objectives: minimizing programmer intervention, presentation of the important aspects of a program, allowing for users to quickly focus, and scalability to nontrivial applications.

There are many other program visualization tools that have had an important impact in the research domain and on the JIVE methodology in particular. *DYNALAB* is a tool that enhances lectures or laboratory exercises with program animation techniques [13]. Although it does not appear to be currently in active development, it was one of the first systems to support reverse execution through a virtual machine. *DDD* is a front-end to `dbx`, and it provides simple list and tree visualizations [94]; however it does not take advantage of the perspective of objects as environments. The "cel" visualizations described by Walker et al. [90] show a high-level model of object-oriented execution. 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; our intent is significantly different despite outward similarities. The drawings described by Hamer [45] show simple and small object diagrams, including useful paradigms for showing collections. However, this work neither addresses diagram scalability nor interactive drawing; it requires offline construction through GraphVis tools.[6]

[6] `http://graphvis.org`

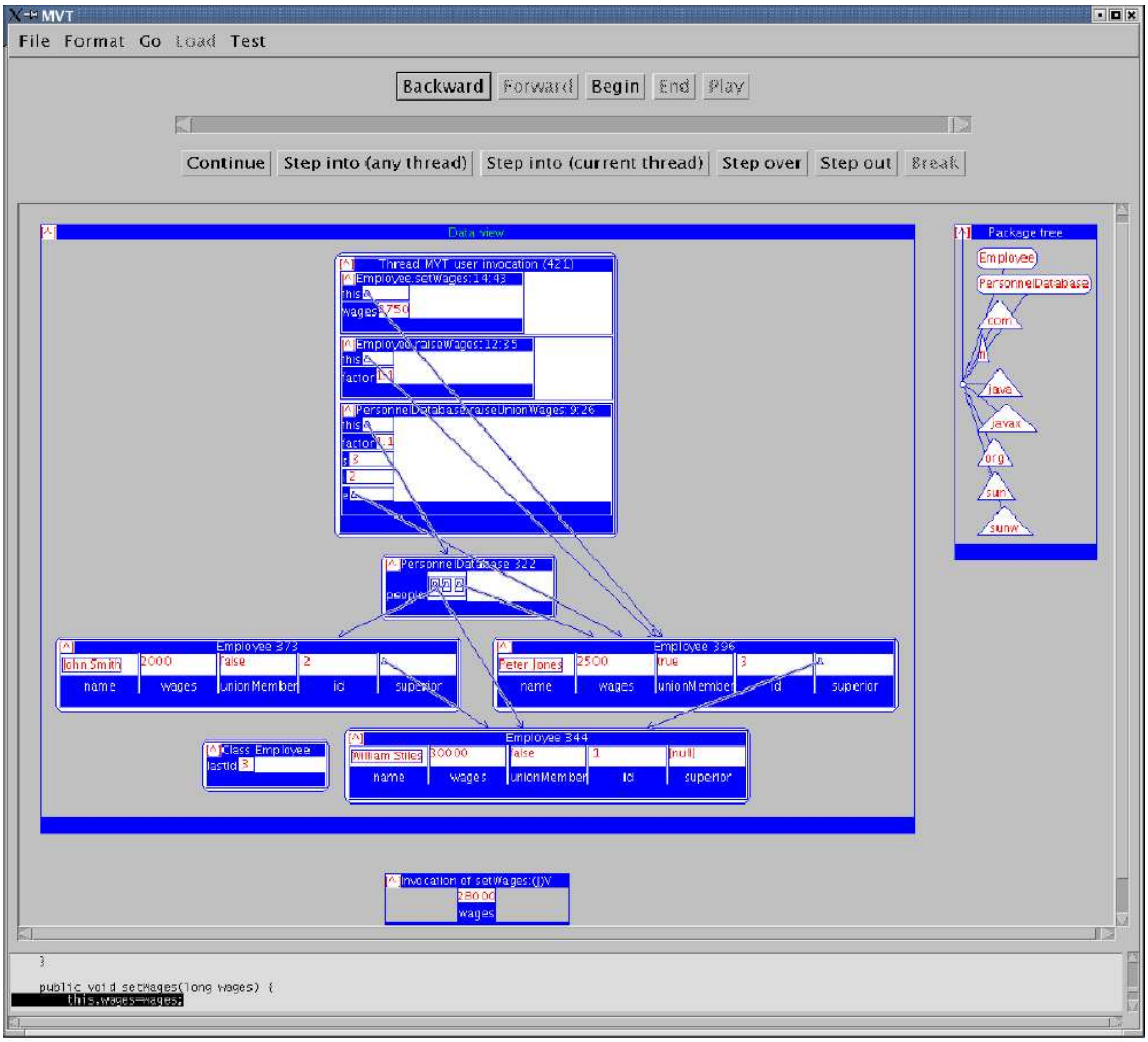


Figure 2.20: A screenshot of Matrix Visual Tester (MVT), a visual debugging aid for Java. MVT features a form of forward- and reverse-execution as well as visualizations of program state. This screenshot illustrates a program state on the left and a view of the package hierarchy on the right.

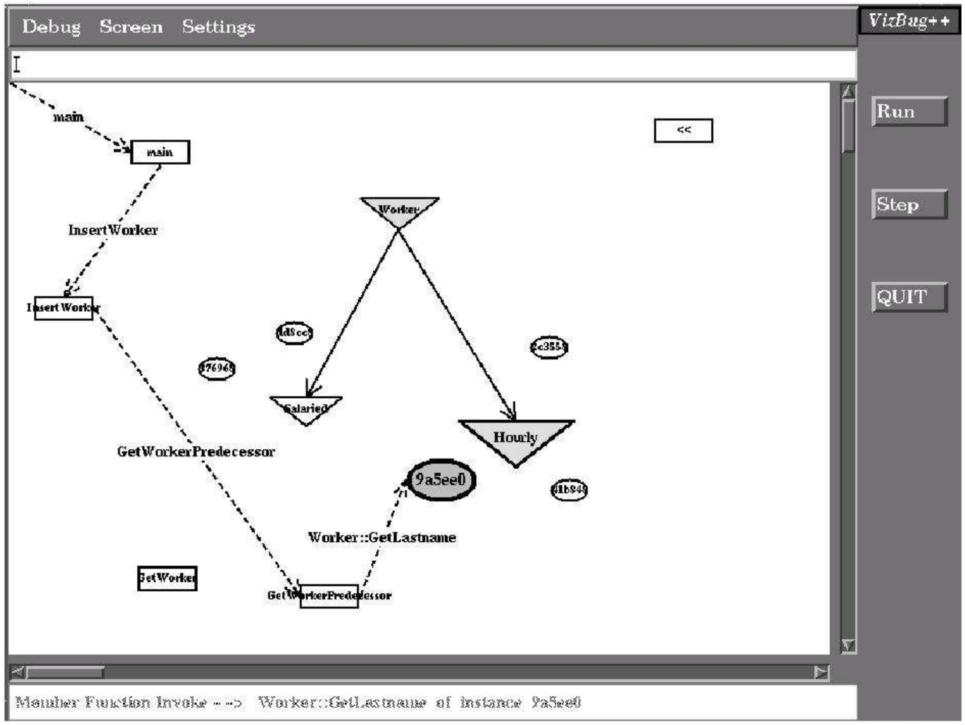


Figure 2.21: A screenshot of the VisBug++ debugger for visual debugging of C++.

Although they are not strictly related to the JIVE methodology, it is worth mentioning a few algorithm animation tools. Algorithm animation research is interested in developing techniques for animating the runtime behavior of specific algorithms, and therefore they tend to be much less broad than the JIVE methodology, although their visualizations can be argued to be more semantically precise for their limited range. JIVE is not an algorithm animation tool, although similar drawing and efficiency concerns emerge. *Zeus* is a classic example of algorithm animation software; it allows users to build algorithm visualizations based on event specifications [18]. *Lens* combines a source-level debugger with algorithm animation into a more integrated visualization environment [69]. Another *Jive* has been developed at Università di Salerno by Umberto Petrillo-Ferraro, and its name is an acronym for *Java Interactive software Visualization Environment.*[7] It is an educational algorithm animation tool for Java, and it features a zoomable user-interface [31]. There are many other algorithm animation tools, and these have been presented to give some concrete examples. An overview of these visualization research through the late 1990s is provided in Wiggins [92].

[7]Petrillo-Ferraro's tool is typeset as "Jive" to avoid confusion; also, note that its acronym expansion is different from our own tool's.

Interactive Execution

Reversible execution has been explored in two general forms: *re-execution* and *state-saving*. The re-execution model, as pioneered in Zelkowitz [93] 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. Another example of this approach is found in the work of Biswas and Mall [14], who use inverse statements to reverse computation at a statement-by-statement level. The alternative is state-saving, a technique that hearkens back to the EXDAMS system by Balzer [7]. 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 approach. This technique has been used in many other works, such as those of West and Panesar [91], who use source-code annotation for fully-automatic incremental state saving.

The state-saving mechanism of the JIVE methodology uses a form of declarative event recording and analysis similar to the work of Richner et al. [80], though the events that are recorded are tailored for integration with Java through the Java Platform Debugger Architecture (JPDA). Our model of recording program execution allows for queries that are not just on specific states, as in the Fox query language [72], 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.

The *time travel* capability of Forms/3 is similar in intent to our model of reverse execution: users are allowed to move forward or backward through time to view the changes in program state [6]. The model of *logical time*, where program entities are perceived as persistent entities whose values change along a time axis, is directly applicable to our visualization approach. When reverse execution changes a value in a program, we will need to develop a model for handling the viscosity problem specific to Java and our implementation. The viscosity problem arises when parts of a program need to be re-evaluated due to changing parameters during reversed execution.

There has been some interesting work in interactive execution in the context of Java. Java's virtual machine architecture provides a convenient environment for implementing interactive execution since the entire state of the virtual machine is readily accessible in a custom-implemented VM. Cook [23] takes advantage of this in his work on reverse execution of Java bytecode. This work involves using a customized Java virtual machine that can store its state: the JVM is stack-based, and this work stores stack frames into separate structures so that they can be restored on command. Cook's work is tightly integrated with the operations of Java bytecode and his custom implementation of them, which does deviate from our own final desideratum for interactive visualization of Java, namely the requirement that standard off-the-shelf Java technology be used for maximum portability and extensibility.

Brörkens and Möller [17] describe the *jassda* framework for generating trace data of program execution. Jassda is not used for interactive execution, but the event recording mechanism is similar to the one used in JIVE. They make the important contribution of developing a technique for obtaining method return values through the JPDA. The JPDA alone does not provide this capability, but by providing a custom classloader, jassda is able to record method return values without source-code modification or compromising Java's and JPDA's cross-platform capabilities. This is made possible through the Apache Byte Code Engineering Library (BCEL).[8]

Graph Drawing

Graph drawing research is mostly independent from the program visualization, interactive execution, and pedagogical research. Essentially, graph drawing refers to the problem of automatically arranging a graph's vertices and edges in order to satisfy a set of aesthetic constraints. One major complication of graph drawing is that these aesthetic constraints are often conflicting. Finding an optimal solution is combinatorially hard, a fact that introduces the need for effective heuristics. Di Battista et al. [24] provide an excellent introduction to graph drawing, covering a broad range of state-of-the-art techniques. Rather than attempt to give adequate coverage to this broad research topic here, we instead investigate a selected set of tools and techniques to show how they relate to JIVE and the desiderata of Chapter 1. A more detailed explanation of the techniques used in JIVE are presented in Chapters 7 through 9.

Layered drawing algorithms have proven useful for drawing both trees and general graphs. A layered drawing assigns vertices to horizontal layers (strata) based on topological ordering. The techniques described by Sugiyama et al. [85] and Reingold and Tilford [76] can be used to produce good drawings according to commonly held aesthetic constraints. The tool *dot* implements a technique for layered drawing that works well on static graphs [33]; Figure 2.4 shows a graph drawing produced by dot. A related tool, *DynaDAG*, uses a similar technique on dynamic graphs [22, 70]; a dynamic graph is a graph that is modified by unit changes such as the addition or removal of a vertex or edge. Figure 2.23 illustrates the dynamic layout made possible through DynaDag. Drawing dynamic graphs is especially challenging since in addition to classical aesthetics of drawing, the algorithms

[8] `http://jakarta.apache.org/bcel`

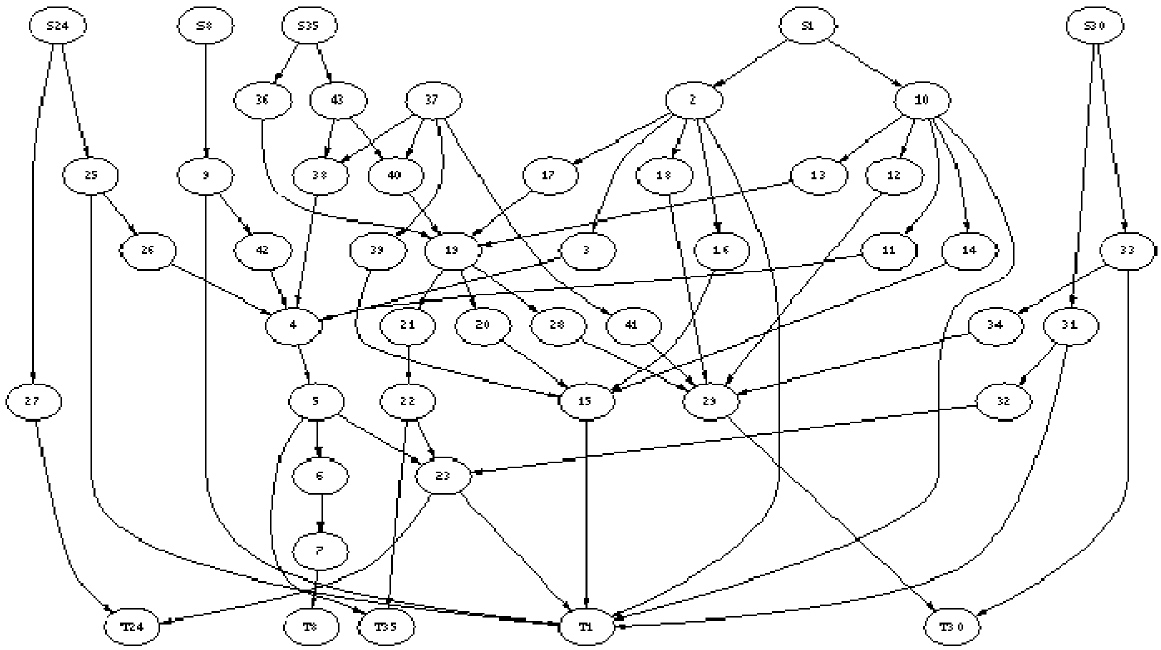


Figure 2.22: A drawing produced by *dot*. This is a hierarchical layered drawing of the graph.

must aim to preserve the *mental map* of the user during the changes [27]. Since JIVE's diagrams are inherently dynamic, as they are intrinsically tied with the interactive execution engine, these techniques of dynamic drawing are applicable to our methodology. However, this is only the case if there is a reversible transformation between JIVE diagrams and mathematical graphs that satisfies the aesthetic constraints.

An alternative to layered hierarchical drawings is to use a force-directed approach, where edges are interpreted as forces acting upon their nodes. *GLIDE* is a system that uses a constrained force-directed approach to optimize interactive drawing [82]. A screenshot of GLIDE is shown in Figure 2.4. One of the benefits of this system is that it allows the user to manually rearrange a drawing, dynamically modifying the rest of the drawing to maintain certain aesthetics (referred to as *Visual Organization Features* in GLIDE). Unfortunately, GLIDE's approach is only practical for use with small graphs, those with less than fifty nodes, which prevents it from satisfying our scalability requirements.

This introduces an important aspect of graph drawing research: for general graphs, general techniques can be used, but drawing diagrams in a specific domain benefits from domain-specific aesthetics. As an example, consider the problem of drawing UML class diagrams, which is similar to yet substantially different from the problem of drawing object diagrams. There has been significant effort in applying class-diagram analysis for drawing UML class diagrams themselves. These efforts

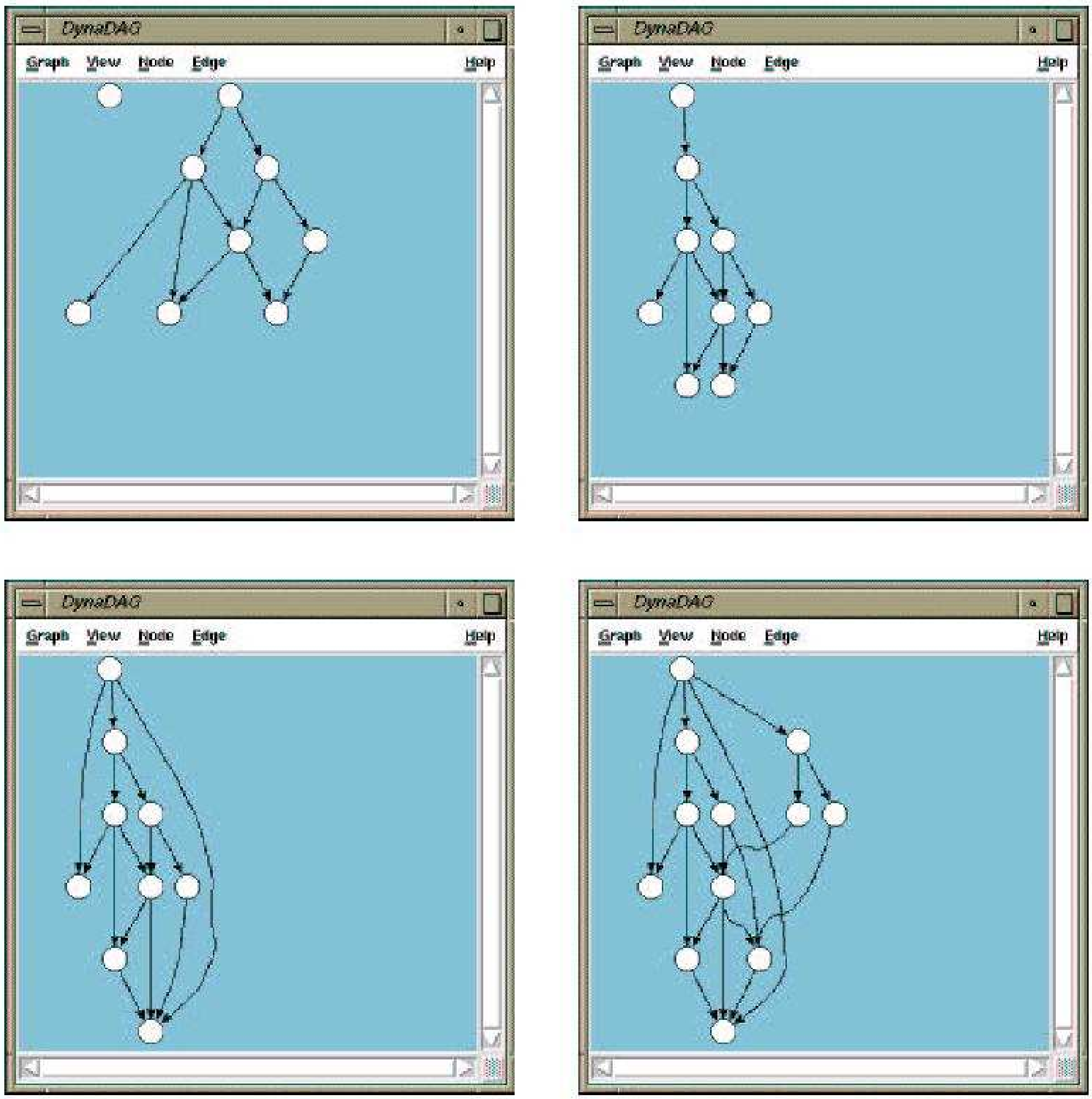


Figure 2.23: Four screenshots of the DynaDAG dynamic graph drawing package. The first and second transitions (reading left-to-right, top-to-bottom) are additions of a single edge. The transition to the last state involves the addition of a subgraph.

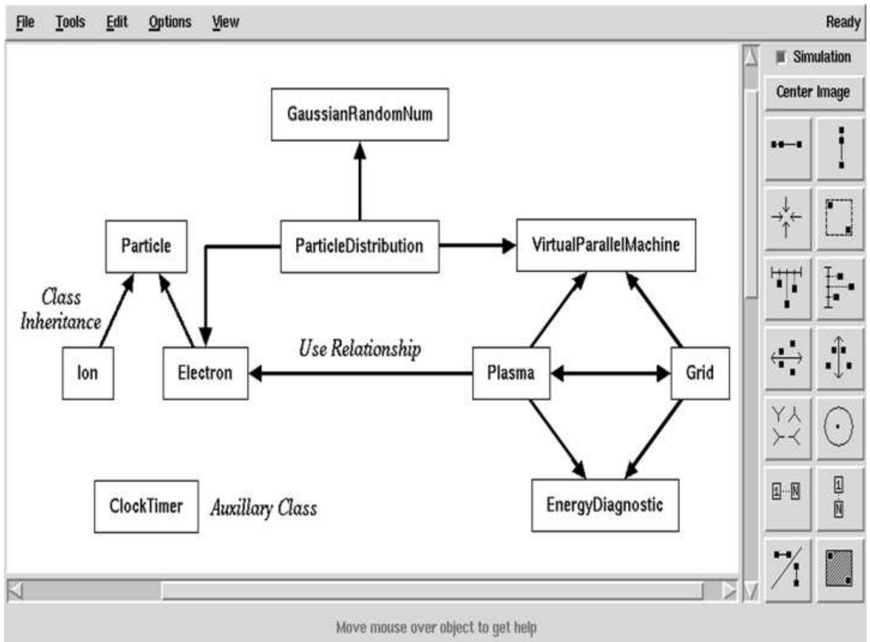


Figure 2.24: A screenshot of GLIDE, a drawing tool that uses a force-directed approach to generate good drawings of graphs.

are founded in the idea that a good drawing of a UML diagram is easier to understand [75]. Although these studies were performed on class diagrams, we expect that similar results would arise for other UML diagram types, including object and sequence diagrams. GoVisual is a UML drawing tool that uses a clustering technique where clusters are formed based on generalization hierarchies [43]. It is one of several works that adopts a topology-shape-metrics to drawing UML class diagrams [30, 24], while other work focuses on using class diagrams as debugging tools [51]. None of these works have used class diagrams as a tool for generating object diagrams. However, there are underlying ideas that can be extracted and applied to JIVE, such as the idea of clustering. Traditionally, clusters are either manually identified [33] or automatically detected in the graph [81]. These techniques are applicable to generic graphs that are domain-independent; the methodology described in Chapter 9 leverages the fact that object graphs are the manifestations of static class relationships.

Chapter 3

Visual Operational Semantics for Java

In the previous chapter, we saw that visual metaphors have proven useful in a wide variety of applications, especially in the domain of software engineering, debugging, and program understanding. This chapter builds upon this strong base of research by contributing a methodology and architecture for visualizing object-oriented programs at runtime, specifically Java programs. In order to visualize Java program states, a notation is needed for depicting these states. We present a *visual operational semantics for Java* to meet this need. In this chapter, the syntax and semantics of our notation are described along with their history and integration with the JIVE methodology.

The term "Java" refers to a programming language, its virtual machine, and its application programming interface (API). The information presented in this chapter is framed such that a general knowledge of Java is required, and so some language-specific issues (such as Java's differentiation between classes and interfaces) are only briefly covered. There are many excellent resources for learning Java, and these should be referenced to clarify any Java-specific uncertainties from this chapter. For example, the *The Java Tutorial* is available for free online at `http://java.sun.com/docs/books/tutorial`. Where applicable, we provide references to relevant sections of the Java Language Specification (JLS) [56] and the Java Virtual Machine Specification (JVM) [64], both of which are necessary references for those seeking a deep understanding of the inner workings of Java.

The remainder of this chapter is structured as follows:

- Section 3.1 describes the desired properties for our visual operational semantics for Java.

- Section 3.2 presents the syntax and semantics for our enhanced object diagrams, which are used to visualize Java program state.

- Section 3.3 explains how our object diagram notation clarifies some of Java's oft-misunderstood operational semantics.
- Section 3.4 presents our definitions for multiple and alternative views of the object diagram, which facilitates more scalable visualizations.
- Section 3.5 presents a mathematical model for representing program state, which allows for a formalization of the notation described in Section 3.2.

3.1 Desired Properties

The previous chapter presented a survey of program visualization techniques. Based on this body of related work and an understanding of program execution and the implications of object-orientation, we present the following desired properties for a graphical notation for Java program states:

1. *Clarify method context.* The notation must clarify the important fact that objects are environments of execution. That is, method activations should be viewed within their proper object contexts. This is necessary in order to convey such ideas as static scoping of variables, behavior of inherited fields and methods, and the effect of calling methods on different objects. This also highlights the significant difference between the behavior of subroutines in object-oriented systems versus pure procedural (*i.e.* non-object-oriented) systems: in a pure procedural context, objects are on the heap and subroutines are invoked on the heap; in an object-oriented environment, the two spaces become inexorably connected as methods are invoked within objects.

2. *Support the Java language.* All of Java's syntactic features must be supported. This includes all language functionality that can be expressed using Java reserved words, but not necessarily all of the possible applications of the expansive Java class libraries and third-party extensions. For example, although desirable, it is not strictly necessary to have a graphical representation for Java Remote Method Invocation (RMI). which is supported through class libraries but not in the Java language syntax itself. The language features for which we mandate graphical support are:

    - classes, objects, and interfaces;
    - fields, methods, and inner and anonymous classes;
    - static variables and "class objects";

- access control modifiers;
- generic and enumerated types;
- and multiple threads with language-level synchronization.

Although multiple threads are only possible through the Java API (specifically, the `java.lang.Thread` class and `java.lang.Runnable` interface), the presence of the `synchronized` keyword brings multithreaded capability to the core of the Java language. Additionally, Java has extensive library support for graphical user interfaces, and any application that uses these is inherently multithreaded: both Swing and AWT use the *AWT-Event thread* to build visual interfaces, process user input, repaint requests, and perform other related tasks. Therefore, support for multithreaded programs is essential to a graphical notation for describing Java's operational semantics.

3. *Planarity.* The notation must be *planar*, meaning that it must be possible to draw object diagrams in two dimensions without the overlapping of object boundaries. More formally, for any three diagram components x, y, and z, it cannot be the case that x is contained in y, y is contained in z, but x is not contained in z. An example is illustrated later in the discussion of Figure 3.2.

   It is not necessary that edges be prevented from crossing, since such crossings are often unavoidable, and existing graph-drawing techniques can be applied to reduce edge crossings. Chapter 7 presents a complete discussion of the issues involved in drawing these diagrams and the nature of edge crossings.

Although there are many techniques for visualization that are useful for their own problem domains, none provide a complete, object-level view of program states that meets all of the criteria outlined above. In the remainder of this chapter, we describe a visual operational semantics for Java that addresses this shortcoming.

3.2 Contour Model Semantics for Java

In 1971, John Johnston introduced the *contour model* semantics for visualizing program execution Johnston [55]. His model can be used to explain the runtime behavior of block-structured programs, and it provided a visual clarification of concepts such as method activation, memory allocation, and static scoping. Examples of Johnston's notation are provided in Section 2.1. Our visual semantics for Java is inspired by this work, and we borrow the name "contour model" despite the important augmentations and additions to the notation. The contour model is used to draw *enhanced object diagrams* (often referenced as *object diagrams* for brevity). As in the classical contour model semantics, the basic compositional unit of our notation is the *contour*. A contour represents an environment for execution, a logical, program-created enclosure that may contain variables, methods, and nested contours. It is important to distinguish between the *contour model* and *object diagrams*. We will occasionally use the two terms interchangeably for convenience, but they refer to two different yet related concepts. The contour model is the abstract model, for representing the operational semantics for Java. An enhanced object diagram is a *contour diagram*, and it is a drawing of a contour model. A single contour model may have many different object diagrams, each focusing on a different aspect of the model or drawn with a different visual paradigm. We also introduce a distinction between a *contour* and a *context*. A reference to a contour is a reference to a specific contour within the contour model. A *context*, like a contour, references a program unit, but not necessarily its contour. That is, there can be a contour for any context, but there does not have to be. Introducing this terminology allows us to refer to those elements of a program that could have a contour but do not necessarily have one.

Jayaraman and Baltus [53] have shown that the application of contour model semantics to object-oriented languages yields two types of contours, object and method contours, where method contours are nested within object contours. This nesting is used to represent the fact that objects are environments of method execution: a method is depicted within its corresponding object. However, significant modifications and clarifications must be made to the classical contour model and the general contour model for object-oriented languages in order to represent Java program execution. One of these modifications is the introduction of a third type of contour, the *static contour*. Hence, there are three types of contours within our contour model:

Object contours represent objects, which are instances of classes. Object contours contain the non-static fields, methods, and inner classes defined in their class. Object contours can be used to represent inner objects.

Static contours represent static space, which is the conceptual location in which static members and static inner objects reside. Object contours contain the static fields, methods, and inner classes defined in their class.

Method contours represent method activations, and they therefore contain the formal parameters and local variables declared within their method definitions.

Each contour in the contour model is identified by a name. For static and object contours, the name is unique within the contour model. Method contour names are unique with respect to their execution context. There is exactly one static contour for every class in the model, but there can be any number of instances of a class and any number of executions of a method. It is therefore convenient to use a numbering system for object and method contours. The following contour naming scheme is consistent with the aforementioned requirements:

- Static contours are named according to the class whose static context they represent. For example, the static contour for the class `java.lang.Object` is identified as "java.lang.Object."

- Object contours are named according to their class and a unique *instance count*, an integer reflecting the relative order in which the instances of the same class were created. For example, the object contour for the first instance of class `java.lang.Object` is identified as "java.lang.Object:1."

- Method contours are named according to their name and an *activation count*, an integer reflecting the relative order in which the method was called with respect to methods of the same name defined in the same context. For example, the method contour representing the first activation of the `main` method of some class `C` is identified as "main:1". The first invocation of a `main` method in a class D could also be labeled as "main:1" since its meaning can be uniquely determined by its name and enclosing context.

Object Contours

An object contour represent an instance of a class. It contains the non-static fields and methods declared within its class, along with two special entries, *this* and *super*. The *this* entry refers to the object itself, and the *super* entry refers to the containing object contour. `Object` instances do not have superclass contours, and hence they do not have `super` fields. In terms of language expressions, the `this` field represents the object referenced by the keyword `this` as used to access fields. Equivalently, it refers to the object accessed through `C.this` within an instance of `C`.

Inheritance is represented in object contours by nesting the contour of a subclass within a contour of its superclass. That is, for a class B which is a subclass of A, an object contour `B:`i must be nested within an object contour for `B:`j, where i and j are arbitrary integers that uniquely identify the instances of `A` and `B`. This implies that, in Java, every object contour will be nested within a contour of type `java.lang.Object` at a depth corresponding to its place in the class hierarchy.

```
class C {
  int x=10;
  static char c = 'J';
  void f() {}
  static void g() {}
}
```

Object:1		
this	Object	Object:1
clone()	method	clone.cf
equals(Object)	method	equals.cf
finalize()	method	finalize.cf
getClass()	method	getClass.cf
hashCode()	method	hashCode.cf
notify()	method	notify.cf
notifyAll()	method	notifyAll.cf
toString()	method	toString.cf
wait()	method	wait.cf
wait(long)	method	wait.cf
wait(long,int)	method	wait.cf

C:1		
x	int	10
f	method	f.cf
this	C	C:1
super	Object	Object:1

Figure 3.1: Sample object contour with its associated source code.

Figure 3.1 presents a sample Java program and a corresponding object contour. The object contour for class C is nested within an object contour for its superclass, which is Java's default base class, `java.lang.Object`. In this example, all of the methods of the superclass are shown, although in practice, such details are often hidden to conserve space and prevent diagrammatic clutter. The implicit `this` field in `C:1` refers to itself; the `this` field in `Object:1` and the `super` field in `C:1` refer to `Object:1`.

Object Contour Groups

A general *contour group* is composed of a contour and all of the contours nested within it. An *object contour group* is a special type of object group that is composed of *virtual* and *concrete* object contours. A *concrete object contour* is an object contour that represents the most specific class of a given type, and a *virtual object contour* is an object contour that represents a concrete object contour's superclass context. An *object contour group* then is a sequence of zero or more virtual object contours followed by a single concrete object contour; it is a sequence of contours that represents a single object context in a program's execution. Virtual and concrete object contours are an abstraction used by the contour model semantics and do not necessarily have an explicit representation in a Java Virtual Machine.

Let $class(v)$ be a function that returns the class of an object contour v. Then formally, an object contour group G is a sequence of object contours $v_1, v_2, \ldots, v_k$ such that: v_{i+1} is nested in v_i and $class(v_{i+1})$ is a subclass of $class(v_i)$ for $1 \leq i \leq n-1$; $v_1, v_2, \ldots, v_{k-1}$ are virtual object contours; and v_k is a concrete object contour. As an example, an instance of `java.lang.String`, a subclass of `java.lang.Object`, could be described in the contour model by an object contour group composed of a virtual object contour for `java.lang.Object` and a concrete object contour for `java.lang.String`.

Static Contours

At first, it may seem reasonable to include static members along with the instance (*i.e.* non-static) members within object contours' member tables. This naïve approach can be partially justified since a single class cannot define static and instance members with the same names, and hence static and instance members are scoped in similar ways. Accessing static variables does not require any special syntax. For example, given a member `x` in a class `C`, `x` can be referenced by name within the context of of `C` regardless of whether it is an instance or static member. However, if static members were listed in an object contour, each object contour would redundantly contain the same static fields. This is undesirable because a change in a static field within one object contour would require distributing that change to every object contour of the same class. Hence it is preferable that each unique field only be shown once in any object diagram.

Since static variables cannot be placed within object contours, a reasonable approach is to declare them outside of object contours. It is desirable to maintain the simple identifier scoping rule of the contour model, namely, that if an identifier is not in the current contour, it must be declared in an enclosing one. Unfortunately, such an approach would necessarily break the planarity restriction as soon as subclassing is introduced, as illustrated in Figure 3.2. That is, there would be unsatisfiable requirements of nesting. Consider as an example the simple program of Listing 3.1. An instance of `B` would need to be nested within the static context of `A` and `B` since it can access both `x` and `y`. However, its superclass object contour would need to be nested within `A` but not within `B`, since it can only access `x`. This nesting requirement is illustrated in Figure 3.2, and it is clearly impossible to satisfy along with planarity, rendering this approach infeasible.

```
abstract class A {
  protected static int x = 10;
}
public class B extends A {
  protected static int y = 20;
}
```

Listing 3.1: Java program with static members

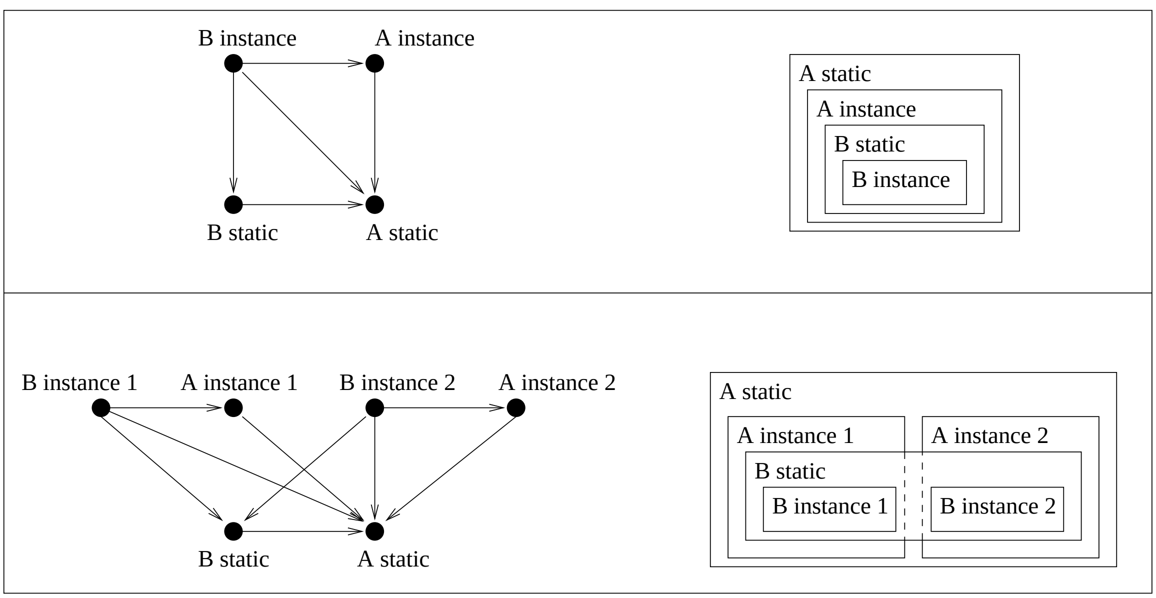


Figure 3.2: Illustrates the impossibility of nesting static contours in the same drawing space as object contours. The graphs on the left indicate the required nesting of graphical components. In the top figure, where there is only one instance of `B`, the nesting is possible, but once more than one instance is introduced (as in the bottom figure), the nesting cannot be satisfied.

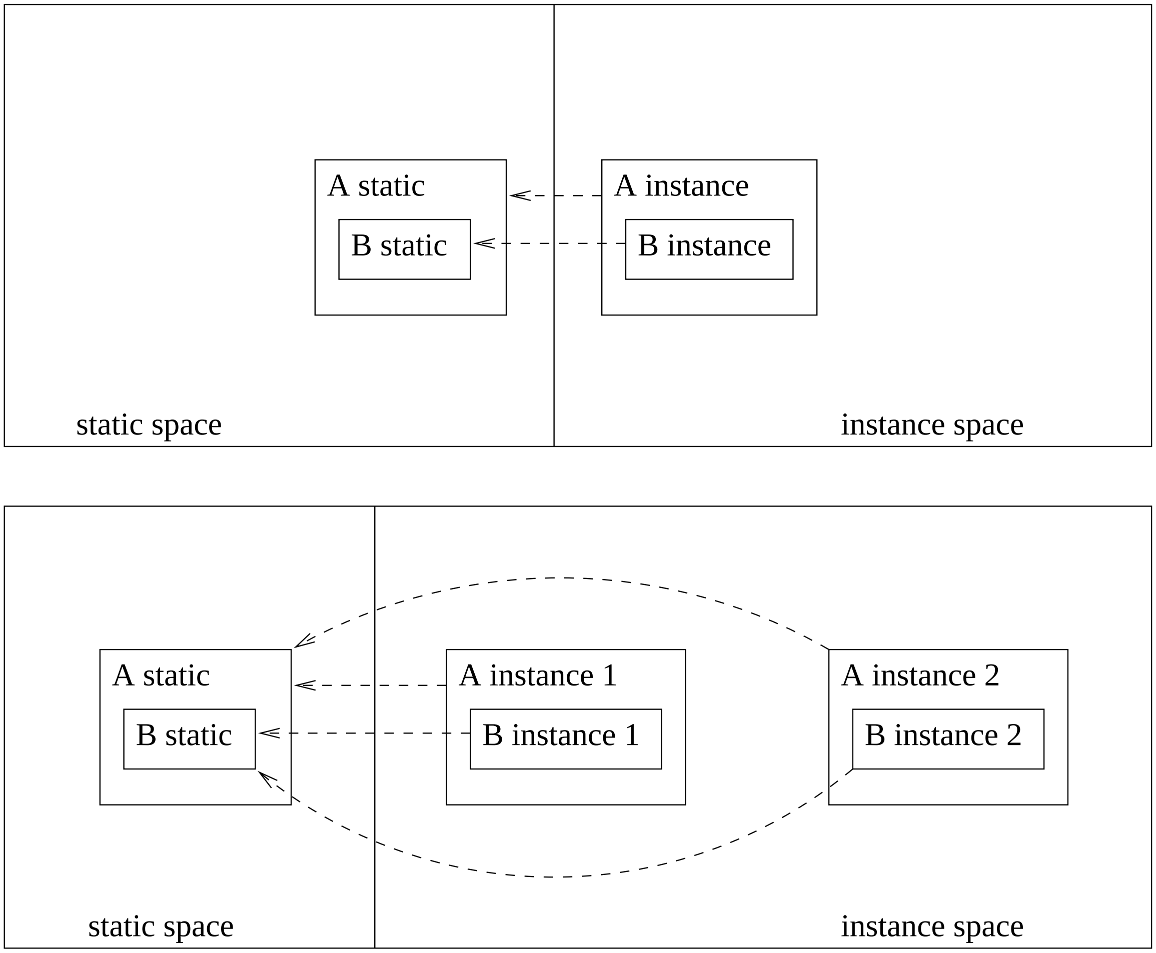


Figure 3.3: The separation of static and instance space. This figure addresses the impossible nesting situation presented in Figure 3.2. The dashed arrows represent *static links*, which connect instances to their static counterparts; these links are generally not shown in the contour diagram since they can be inferred from our contour naming scheme.

Our solution to this problem is the separation of *instance space* (or *object space*) from *static space* [37]. The static contour for a class is introduced in static space, and it contains the static members of that class. Object contours, representing the instances of classes, are placed in object space. This concept is illustrated in Figure 3.3, which addresses the same situation depicted in Figure 3.2. These two spaces are conceptual and part of the contour model, and they are not necessarily graphically separated in a contour diagram. Static contours, like object contours, are nested according to their inheritance hierarchy. Given a class `C` that extends class `B`, the static contour `C` is nested within the static contour `B`. Using this technique, when a static variable changes value, it changes in only one place. The primary drawback is that there is a visual separation between the object and static space. Figure 3.4 is a static contour for the same program considered in Figure 3.1. There are no members in the superclass contour because `Object` defines no static

```
class C {
  int x = 10;
  static char c = 'J';
  void f() {}
  static void g() {}
}
```

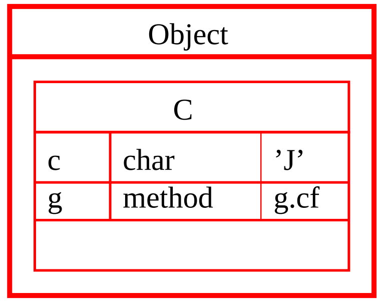


Figure 3.4: Sample static contour.

members, and the empty member table is entirely hidden. The member table of `C` contains entries for the two static members of the class C. As shown in Figure 3.3, we define *static links* that connect instance contours to their static counterparts; however, since these can be directly inferred from our contour naming technique, and since static space is very limited in size compared to object space, static links are rarely shown in contour diagrams.

Generally, we omit from a diagram any details which do not help convey the meaning intended for it. This is analogous to using diagrams to communicate system design, where it is advantageous to have different diagrams of the same system, each with different focus and granularity of detail. The object diagrams we present are scalable in the same sense that the UML is: different views of the same data can produce diagrams with different expressive foci. In practice, static links and static space are often omitted entirely from a diagram. Many programs only enter static space during initialization through the static `main` method or for access to constants. Similarly, it is redundant and space-consuming to show all of the methods defined in every instance of a class, or to show the complete details of every superclass contour. Hence, these details are often omitted from view.

Method Contours

A method contour represents the activation and execution of a method. This should not be confused with an entry for a method in a member table, which represents the definition of a method. A method contour has a member table, but unlike object and static contours, this member table may not contain method definitions because Java does not allow the definition of methods within methods (*cf.* Section 3.2). Thus, method contours cannot contain nested method contours. Furthermore, object contours cannot be nested within method contours, as this would imply that the object is contained within and dependent upon the lifecycle of the method; the contour model semantics for anonymous inner objects, which can be created within methods, are presented with the discussion of inner objects later in this chapter.

```
class Test {
  public static void main(String[] args) {
    Test t = new Test();
    t.sayHello();
  }
  void sayHello() {
    System.out.println("Hello, World.");
  }
}
```

Listing 3.2: Java program used to illustrate method contours in Figure 3.5

The member table of a method contour contains the variables defined within the method, including both formal parameters and local variables. A method contour's member table contains one special entry, the *return point and dynamic link*, abbreviated *rpdl*. The function of this field is to indicate the context to which control returns when this method exits. Given a method contour `f:n` in a contour `C:`m that calls some method `g` in line four of its source program, then the *rpdl* of the contour for `g` would be "line 4 in C:m.f:n." If a method is the first on its thread and was therefore called directly by the Java Virtual Machine, then the reserved value "System" is listed as the value of the *rpdl*. Threads are not otherwise labeled, since the name of a Java thread is mutable, but the specific thread on which a method is executing is specific implicitly by the chain of dynamic links. In practice, we have found it useful to color-code each thread. This is especially true in programs with graphical user-interfaces, where the GUI is usually built on the *main* thread, but event processing occurs on the *AWT-Event* thread. The rpdl can be represented as a text label or as an arrow that points to the calling context; in this latter case, we refer to the arrow as a *method return link* or simply *return link*.

Method contours are always placed in the context in which they are defined. This is the model's representation of static scoping in Java; a language with dynamic scoping would instead create the method contour where it is called. A corollary is that method contours for static methods will be nested within static contours, and non-static method contours will be nested within object contours. This policy places method contours within the contours whose member tables contain the method's definition.

The program of Listing 3.2 can be used to illustrate the behavior of method contours. An object diagram is given for this example in Figure 3.5. The method contour for `main:1` is nested within the static contour `Test`, and the method contour for `sayHello:1` is within the lone object contour, `Test:1`. This diagram also illustrates how structural links and dynamic links can be expressed visually as arrows instead of as text in the member table. Towards the bottom of the diagram is

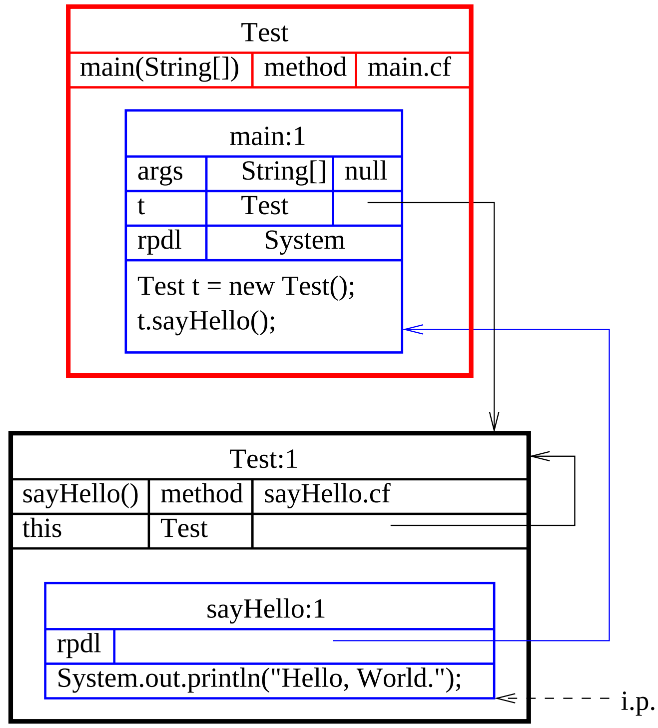


Figure 3.5: "Hello, World" program to illustrate method contours.

the *instruction pointer*, *i.p.*, which is an important part of program state visualizations that contain method contours. The instruction pointer indicates the current location of execution; in the case of Figure 3.5, it indicates that the program has just executed the line that contains the `println` method call.

Method contours are normally removed from the contour diagram when they have terminated, but they are not actually removed from the conceptual contour model. However, it keeps diagrams simple to remove methods since Java does not support re-entry of expired methods. However, there are cases where it is preferable to shade returned method contours rather than to remove them; this technique is demonstrated in the simple program of Figure 3.6. The shading approach is usually followed if the drawings are drawn by hand, since it allows the reader to trace the execution sequence. That is, one can easily see what has already happened in program execution as it relates to expired method activations. Another benefit of this approach there is no need to repeatedly erase large portions of the diagram, which means that the diagram is strictly expanding rather than expanding and contracting during execution. However, when considering integration into an environment for interactive visualization such as JIVE, the benefit of showing stale method contours is eclipsed by their inordinate use of screen space. Furthermore, the JIVE methodology incorporates a visualization of execution history along with views of program state, and hence the visualization

```
class Utils {
  public int factorial(int n) {
    if (n<=0) return 1;
    else return n * factorial(n-1);
  }
}
```

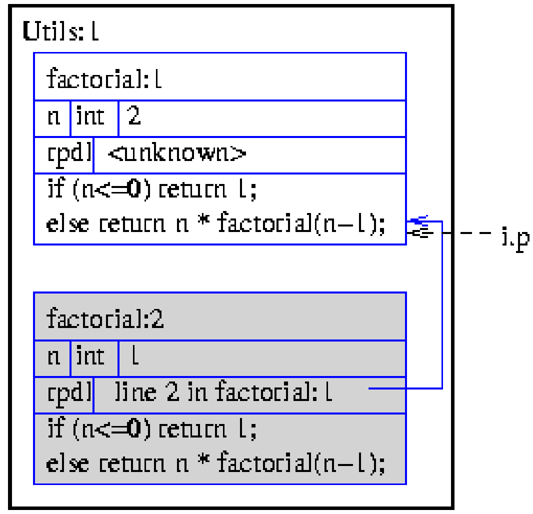


Figure 3.6: Illustration of method shading rather than method removal. Normally, returned methods are removed from the model; this figure shows how they can be shaded instead in order to show that they had existed but are now defunct. It is impossible in the contour model to re-enter the contour for a returned method.

of expired methods is redundant in such a system.

Constructors are used to initialize objects upon their creation. These methods work in tandem with instance initializers and instance member variable initializers in order to ensure that an object is correctly created. Constructor declarations are not members according to the Java Virtual Machine specification (JVM §2.12), and hence they are not shown in the member table. Instead, the invocation of the constructor and of appropriate initializers is handled by a special method, `<init>` (JVM §3.9). Hence, although there are no entries for constructors or initializers in the member table, the creation of an object coincides with the creation of a method contour `<init>` within it. When a class is loaded, a static initializer ensures that all of the static members are properly initialized. This initialization is accompanied by the invocation of a virtual static method `<clinit>` (JVM §3.9). An example is shown in Figure 3.7, in which the source code for constructors is shown in the initializer method contour.

Object References

Java distinguishes between *primitive* and *reference* types. The primitive types are the language-specified data types such as `int` and `char`, and the reference types correspond to classes and are either user-defined or supplied by the Java API. Hence, classes such as `java.lang.Object` and `java.awt.Frame` are reference types. Similarly, variables within the contour model for Java will be of either primitive or reference types. A primitive type is encoded as a string version of its literal;

```
class InitExample {
  C c;
  static {
    c = new C();
  }
}
class C {
  int x = 10;
  int y;
  {
    y = 15;
  }
}
```

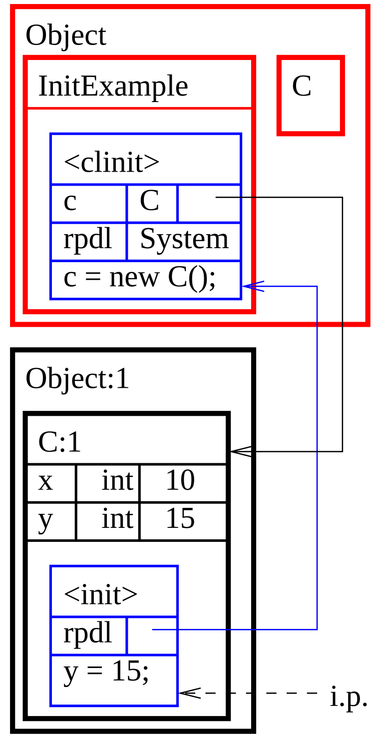


Figure 3.7: Example of an instance initializer (`<init>`) and a static initializer (`<clinit>`).

for example, an *int* variable's value can be expressed as the Java integer literal "10." A reference type is a reference to an object, and so its value in the contour model is a *structural link* to the corresponding object's contour. For example, the value of a `Tree` variable in Java can be expressed in the contour model as "Tree:5," where `Tree:5` is an object contour. Like return links, structural links can be specified as text or as a graphical edge connecting the variable to its contour. Reference variables without value are shown in a member table as having the value "`null`."

The manner in which structural links' endpoints are handled reflects the variable scoping semantics of Java. A non-null object reference member always points to an object of the same type as the member's type. For instance, given a field `x` with type `C`, the contour model value for `x` must either be null or a reference an object contour whose type `C`. According to subtype polymorphism, it is possible that the actual class of the referenced object is a subclass of the member's type. For example, `x` could be assigned an object whose type is `D`, where `D` is a subclass of `C`; however, in the contour model, `x` will still refer to the `C` object contour, not the `D` object contour. We use the term "class" loosely here to mean Java classes as well as Java interfaces. For example, a field `y` with type `I`, where `I` is an interface, must either be null or reference an object contour whose class (or superclass) implements `I` or a subinterface thereof.

```
class A { int n = 10; }
class B extends A { int n = 20; }
class Driver {
 public static void main(String[] args) {
   A a = new A();
   B b = new B();
   A x = new B();
   System.out.printf("%d %d %d", a.n, b.n, x.n);
 }
}
```

Listing 3.3: Java program illustrating static scoping

As an example, consider the Java program of Listing 3.3. This program will print the sequence of integers "`10 20 10`". The fields `a` and `b` clearly reference objects of type `A` and `B` respectively. However, `x` references an instance of `B` through its supertype `A`. Hence, the value given for `x.n` is the value of `n` as initialized in `A`, not `B`. This semantics is clarified using the technique exemplified in Figure 3.8, which shows an object diagram for the above program. While `a` references `A:1` and `b` references `B:1`, `x` references `A:2`, an object contour for `A`, since the type of `x` is `A`. Java uses a shadowing semantics for data members but an overriding semantics for method members, and so this example would be quite different if the referenced member were a method. The contour model semantics for this overriding and shadowing is discussed in more detail in the presentation of Java's handling of `this` and `super` in Section 3.3.

Block Expressions

The representation of method activations as contours follows from the observation that objects are environments of program execution. A contour represents a program unit, and a method is clearly a program unit that can be represented as such. A method may reference its own local variables or the fields of its execution context (or of the superclass context, *etc.*). Java allows for block expressions of arbitrary depth, each of which may contain local variable definitions, and we use a similar contour model semantics for these.

Each block expression is conceptually a nameless function, and block expressions are therefore like lambda expressions. A block expression inherits the environment of its enclosing method and adds its own local variables to it. Hence, we represent block expressions with *block contours*, which are essentially nameless method contours. A block contour always executes on the same thread as its enclosing contour. Names are not required for these block contours since they cannot be explicitly referenced within Java; this contrasts with languages such as Smalltalk, in which arbitrary block

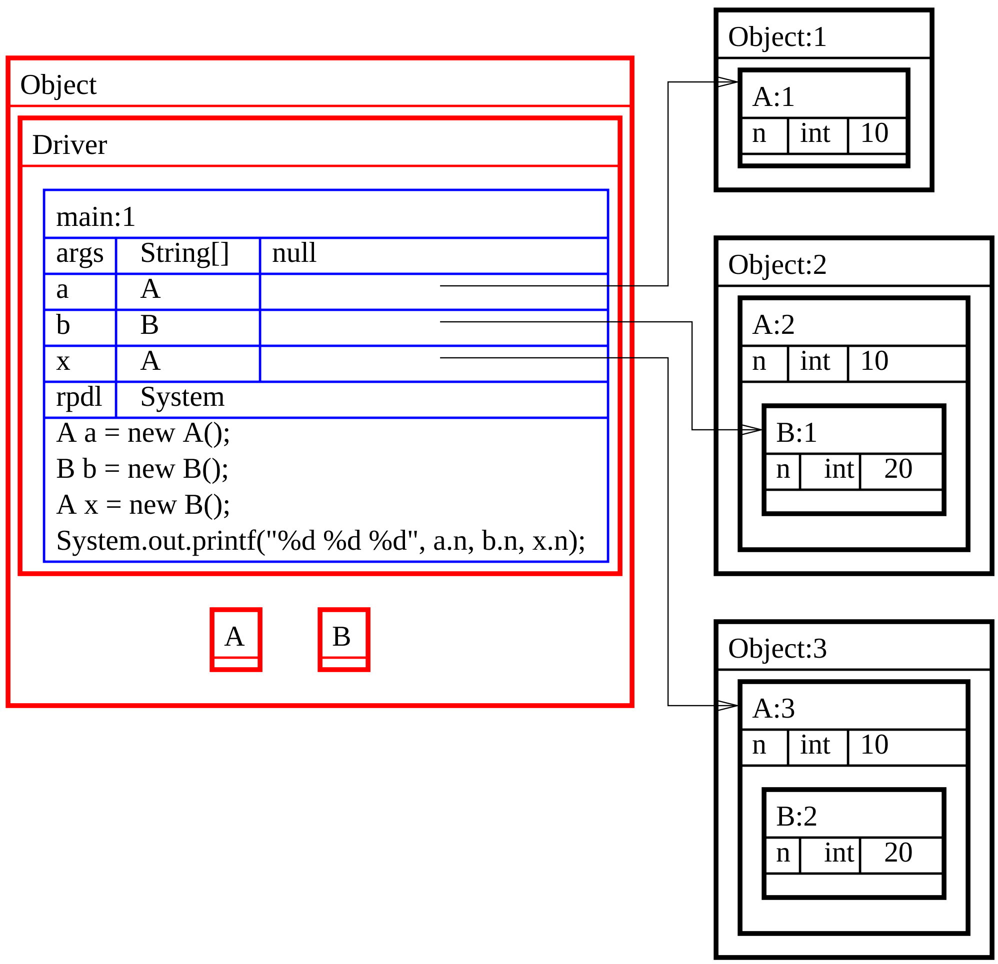


Figure 3.8: Object diagram to clarify how structural links point to object contours according to their declared class, not their actual class.

```
public class BlockExample {
  public static void main(String[] args) {
    int x;
    if (args.length < 1) {
      double d = Math.random();
      System.out.println("Using random argument.");
      x = (int)d;
    }
    else {
      System.out.println("Using provided argument.");
      x = Integer.parseInt(args[0]);
    }
  }
}
```

Listing 3.4: A Java program that contains block expressions. A contour diagram for this program is shown in Figure 3.9.

```
public void m() {
  {
    int i = 100;
  }
  {
    Integer i = new Integer(100);
  }
}
```

Listing 3.5: Multiply defined variables in parallel blocks

expressions are objects and can be referenced, lazily evaluated, or passed as arguments to other methods [40]. Since each block may contain an arbitrary number of nested blocks, block contours may be nested within each other to any depth, and since each block has exactly one immediate parent, block expressions satisfy our planarity requirement in the contour model. Figure 3.9 provides a sample block contour, representing the program shown in Listing 3.4.

In practice, we frequently incorporate block contours into method contours. Java does not allow any shadowing within block expressions with respect to their containing method; that is, if a method contains a definition of a field f, then no nested block expression may define a field f. However, this is not sufficient to eliminate block contours altogether, since parallel blocks (*i.e.* non-nested blocks within the same method) may define local variables with the same name but different types. The program of Listing 3.5, for example, demonstrates a syntactically valid Java program in which the variable `i` is defined twice within one method, using different types in each block. Java is statically and strongly typed, and so it is not possible for any variable reference to be ambiguously scoped. However, the possibility presented in Listing 3.5 demonstrates that block contours are still necessary,

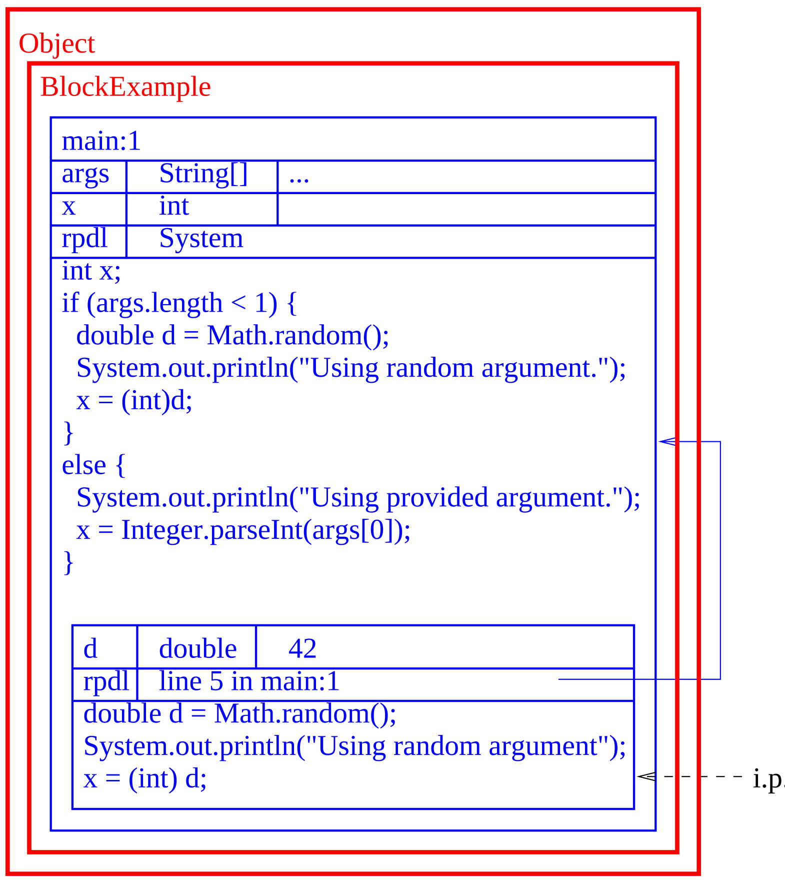


Figure 3.9: An example of a block contour. The block contour here, shown nested within `main:1`, represents the block expression that is executed if the conditional is true.

```
class Example {
  static Runnable runnable = new Runnable() {
    public void run() {
      System.out.println("Anonymous static inner class.");
    }
  };
  static class NamedStaticInner {
    public static void print() {
      System.out.println("Named static inner class.");
    }
  }
}
```

Listing 3.6: Declarations of static inner classes. `NamedStaticInner` and the anonymous inner class implementing `Runnable` are both static inner classes.

since it would invalidate contour model semantics to include two definitions of `i` within the method contour for `m`, and it would likewise invalidate the semantics to define `i` once with two types (*e.g.* `int` and `Integer`). Therefore, block contours are necessary in order to define a valid contour model semantics for Java.

Inner Classes

Java supports the definition of classes within other classes or even within methods; these are called *inner classes*, and they were introduced in version 1.1 of the Java language. An inner class' *enclosing class* is the class that contains the definition of the inner class. It is possible for an inner class' enclosing class to also be an inner class. *Anonymous classes* are nameless instance inner classes; we identify these in the manner used by Java compilers, by assigning them unique labels derived from their enclosing class' name and an integer index. Specifically, given a class C that defines n anonymous classes, we identify these anonymous classes by assigning them integer indexes according to their location in the Java source file; the names used for these classes are $C\$1$, $C\$2$, ..., $C\$n$, respective to their relative positions in the file. Besides this naming convention, anonymous classes can be used as a named inner class in the contour model in all respects.

We distinguish between two types of inner classes: *static inner classes* and *instance inner classes.* This distinction is somewhat implicit within the Java language specification, but it is necessary to explicitly differentiate between static and non-static (*i.e.* instance) inner classes in the contour model. A static inner class is an inner class that is either defined with the `static` keyword or is an anonymous inner class whose instance is assigned to a static field. Examples of static inner classes are shown in Listing 3.6. A static inner class definition may contain both static and instance

```
class Example2 {

  private Runnable runnable2 = new Runnable() {
    public void run() {
      System.out.println("Anonymous instance inner class.");
    }
  };

  class NamedInstanceInner {
    public void print() {
      System.out.println("Named instance inner class.");
    }
  }
}
```

Listing 3.7: Declaration of instance inner classes. `NamedInstanceInner` and the anonymous class used by `runnable2` are both instance inner classes.

members. Correspondingly, a static inner class has a static contour in the contour model, and this contour is created when the class is loaded. Static inner class instances (*static inner objects*) can access their own static and instance members, as well as any inherited static members. However, static inner objects do not have enclosing objects, and so both the static and the object contours for a static inner class are created in static space. Specifically, a static inner class' static contour is placed within its enclosing class' static contour, and static inner object contours are placed within their class' static contour. A complete example is given following the discussion of instance inner classes.

The set of instance inner classes is comprised of the set of non-static named inner classes and the set of anonymous classes whose instances are assigned to non-static fields. Listing 3.7 provides an example of instance inner class definitions. An instance of an instance inner class (an *instance inner object*) is always contained within an *enclosing object*. Instance inner classes may not declare any static members, and hence there are no static contours for instance inner classes. Object contours for instance inner classes are created and nested within their enclosing objects. An object of an instance inner class `I` defined within a class `C` can explicitly reference its enclosing instance using the syntax `C.this`; this is reflected in the member table by including a `C.this` member in each instance of `I`.

Listing 3.8 presents a short program containing both instance and static inner classes. Figure 3.10 provides an enhanced object diagram showing the ending state of the program's execution. This program illustrates the use of both static and instance inner classes. Notice that the variable `min` referenced in the initialization of `StaticInner` is the one defined within `StaticInner`, whereas the

```
class Enclosing {

  static int min = 0;

  static class StaticInner {
    static int min = 0;
    int instance = min++;
  }

  class InstanceInner {
    int instance = min++;
  }

  InstanceInner ii1, ii2;

  static StaticInner si1, si2;

  Enclosing() {
    ii1 = new InstanceInner();
    ii2 = new InstanceInner();
  }

  public static void main(String[] args) {
    si1 = new StaticInner();
    si2 = new StaticInner();
    Enclosing e = new Enclosing();
  }
}
```

Listing 3.8: Program containing static and instance inner classes.

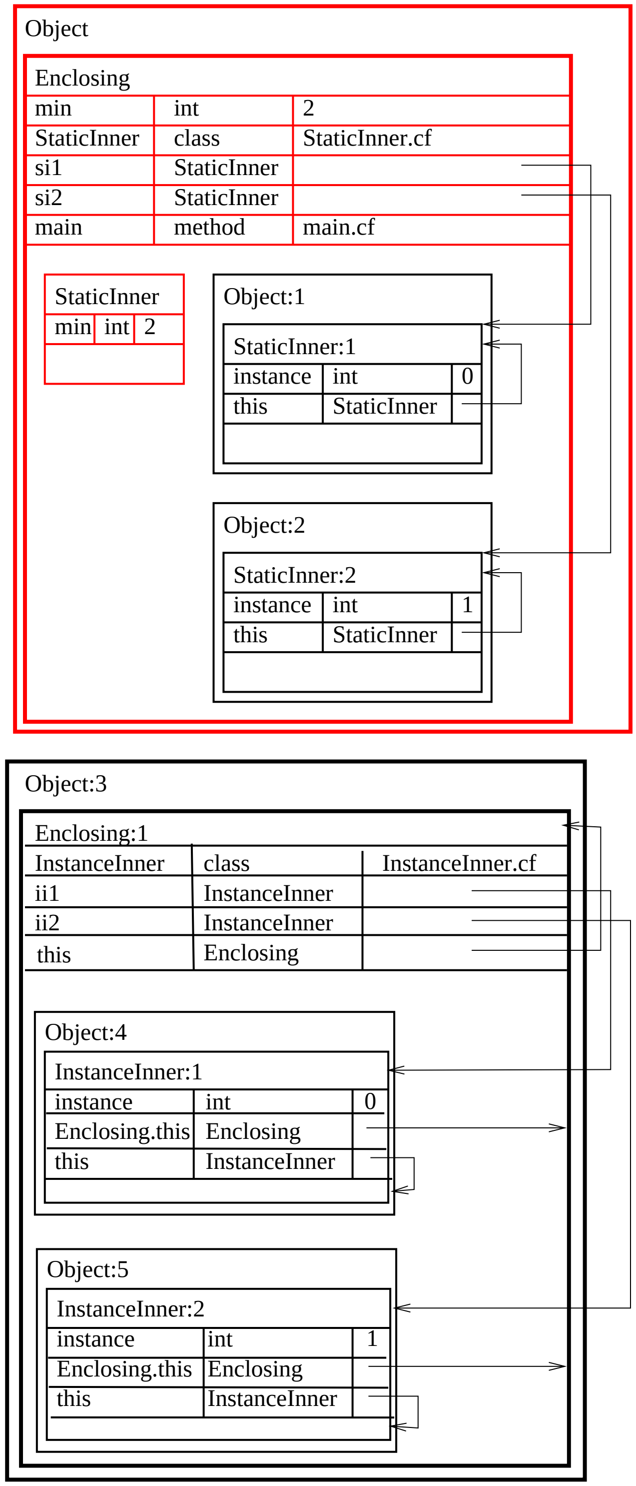


Figure 3.10: Example of inner classes. Static contours are shown in red, instance contours in black. The static inner class `StaticInner` has one static contour, `StaticInner`, and two instance contours, `StaticInner:1` and `StaticInner:2`. The instance inner class `InstanceInner` has no static counterpart, but it does have two instances, `InstanceInner:1` and `InstanceInner:2`.

```
class ExceptionTest {

 public static int sum(int[] a) {
  if (a==null || a.length==0)
   throw new IllegalArgumentException();
  else {
   int total=0;
   for (int i=0; i<a.length; i++)
    total+=a[i];
   return total;
  }
 }

 public static void main(String[] args) {
  try {
   sum(null);
  } catch (IllegalArgumentException e) {
   e.printStackTrace();
  }
 }
}
```

Listing 3.9: Program with exceptions

one referenced in the initialization of `InstanceInner` is the one in `Enclosing`. Both variables have ending values of 2, corresponding to the number of instances of each inner class.

Exceptions

Exceptions provide a language mechanism for disciplined jumps between locations in source code. In the object-oriented world of Java, exceptions are themselves first-class objects. Although they exhibit special behavior in their being thrown and caught due to their extending `java.lang.Throwable`, exceptions are normal objects. The flow of control around the throwing and catching of exceptions is defined by the language specification, but in terms of the object structure, exceptions are first-class objects like any other [41]. As an example, consider the program shown in Listing 3.9. A contour diagram for this program is given in Figure 3.11. As the figure shows, the exception is simply an object, originally created in the `sum:1` method, now referenced by the variable `e` in `main:1`.

In the contour model, there is no special behavior for exception objects. The only consideration in the contour model is that when an exception is thrown in one context and caught in another, method contours can be unexpectedly exited. That is, a method contour may return before it encounters a `return` statement or the end of its block. Furthermore, the location of return is not the one specified in the method contour's `rpdl`; instead, it is the location of the appropriate `catch`

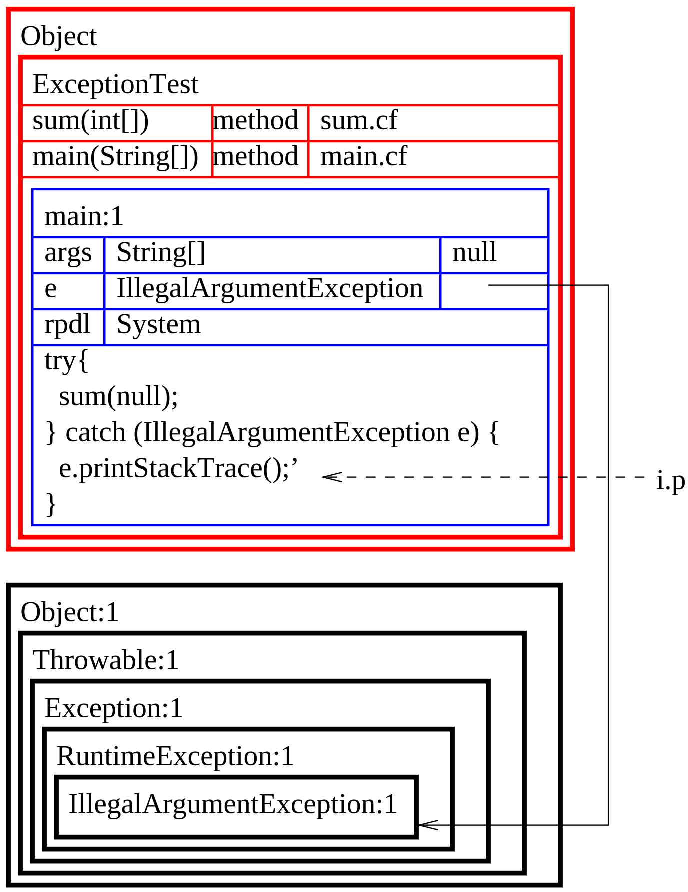


Figure 3.11: A Java Exception in a contour model. The exception is a first-class object, like any other, referenced by the local variable declared in the `catch` clause.

statement.

Interfaces

In Java, an *interface* provides method signatures without giving implementations for them. All concrete classes that implement the interface must provide implementations for all the methods of the interface, either themselves or in their superclasses. Java's interfaces provide a syntactic mechanism for *interface inheritance*, whereas its class `extends` mechanism supports *implementation inheritance*. If interfaces were restricted to only defining method signatures, then there would be no need to represent them in the object diagram; this would restrict them to design-level specification, with no necessary contour model representation. Structural references to interface types could point to the object contour whose class implements the interface without loss of generality, and, for clarity, object contours could optionally be annotated with the interfaces they implement. Unfortunately, this simple approach will not work since interfaces are in fact not restricted to only declaring method signatures.

The Java specification for interfaces allows them to define constants[1] in addition methods signatures. In fact, all fields declared in an interface are necessarily public, static, and final (JLS §9.3). These constants must exist in some space in the contour model, but a similar complication is encountered here as in the definition of static members. The fields are shared among all classes that implement their interface, and so it is undesirable to show these in each corresponding object contour. However, the fields cannot be simply placed into the static context classes that implement their interface since this can still lead to redundant representation of the constants: an interface can be implemented by any number of classes, including zero. Just as it is undesirable to show static members in every object contour of the class, it is undesirable to show interface constants in every static contour whose class implements the interface.

A reasonable approach to circumventing this problem is to introduce *interface contours* in static space, but these would be separate from the static contour for `java.lang.Object`. For each interface `I` that defines constants, an interface contour `I` is created. The interface contour for each interface `J` that extends interface `I` would then be nested within the interface contour of the superinterface; this is necessary because constants defined in an interface hide any constants of the same name that are defined in superinterfaces (JLS §9.3). Unfortunately, this simple approach is not sufficient since interfaces are not bound by a single-inheritance rule as classes are. Hence, this approach will not

[1] Although Java reserves the keyword `const`, there is no associated semantics for it. A `static final` member in Java is as close to a C-style constants as possible in Java.

work since it breaks our planarity requirement.

The problem introduced by constants defined in interfaces exposes a fundamental representational limitation of the contour model semantics for object-oriented languages, namely that it is impossible to represent any form of multiple inheritance within two-dimensional space. Any time multiple inheritance is allowed, unsatisfiable topological constraints can emerge. However, this is not an insurmountable problem in visualizing Java execution states since this multiple inheritance is restricted and disciplined, since it is restricted to interfaces. Our solution is inspired by the fact that, given a class that implements an interface, the methods from the interface are treated as normal class methods. To see which methods are defined in which interface, a programmer would refer to the program source code or associated documentation. Our methodology for constants defined in interfaces, then, is to simply not represent them in the contour model visualization at all. Each of these constants is not only defined in the interface but also necessarily initialized therein. Hence, a reader who wishes to determine the value of a constant may simply refer to the program source code. This is facilitated in environments such as JIVE that allow for concurrent visualization of the object diagram and the program source code.

This solution strategy is not optimal for expressing the full operational semantics of Java specifically because of Java's lack of proper constants. For example, an interface constant can reference an object, but that object's state can change as any other object can. However, it is important to realize that the primary issue is a tradeoff between expressiveness of the model and maintenance of planarity. Since contour model planarity is a requirement of our representation, it is preferable to use this elision model rather than break the visual continuity of the contour diagrams.

Strings

Strings in Java are first-class objects, but they are unique in the Java API in that there are syntactic operators for manipulating them. The operators `+` and `+=` can be used for string concatenation. At the level of the compiler and virtual machine, the concatenation is actually carried out by a series of calls to the `append` method of the `StringBuffer` class since `String` objects are immutable. Another syntactic consideration is that `String` objects can be created from literals, and this gives them a similarity to primitive types that other classes do not share.

It is possible to visualize strings as ordinary objects, in which case the contents of a string would most likely be represented in a character array, although the implementation of `String` is up to the library implementation. However, the motivation for our notation is to understand programs and

```
class AliasString {
 public static void main(String[] args) {
  String s1 = "ba";
  String s2 = "na";
  String s3 = s2;
  String s4 = s1+s2+s3;
  System.out.println(s4);
 }
}
```

Listing 3.10: Java program illustrating string aliasing. A contour diagram for this program is provided in Figure 3.12.

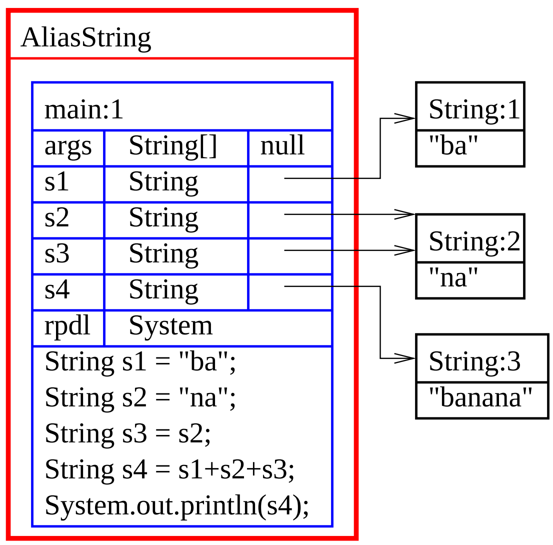


Figure 3.12: String representation and aliasing in an object diagram. `String:2` is references as both `s2` and `s3`.

the semantics of object orientation, not to understand the implementation of Java virtual machines. Hence, our methodology is to show `String` object contours as first-class object contours whose contents is a string literal. The value of this representation can be expressed through an example. Since Java strings are first-class objects, there can be any number of references to them. Hence, aliasing can be a problem [3]; this is one of the reasons for the `intern` method of the `String` class, a method that returns a unique canonical representation for a string. The program in Listing 3.10 illustrates how strings can be aliased, and Figure 3.12 shows how our notation clarifies this aliasing.

Care must be taken when comparing strings in Java since the expected behavior based on source code sometimes differs from the actual library and virtual machine implementation. A pool of strings is maintained by the `String` class, and it is not directly accessible. Whenever a new `String` object is requested, if an equivalent one is in the pool, the pooled string is returned instead. Hence, there

```
public static void main(String[] args) {
 String s1 = "same";
 String s2 = "same";
 System.out.printf("%b %b", s1==s2, s1.equals(s2));
}
```

Listing 3.11: Java method using two identical strings. The output of this method is “true true.”

is a single canonical representation for each logical string. The program segment of Listing 3.11 illustrates this property of Java strings. One who is accustomed to object-orientation but not to Java might expect the output to be “false true” since it appears that `s1` and `s2` refer to two different `String` objects, but in fact, they refer to the same canonical representation. The `String` class, without programmer intervention, has re-used the same `String` object for both `s1` and `s2`. Hence, the output of this program is “true true.”

Arrays

Arrays are Java’s only fully-language-integrated aggregate. Java 5.0 has brought the Collections API closer to the language level with the syntactic integration of generics and an enhanced `for` loop; however, the Collections classes still must be accessed through the `java.util` package and their functionality obtained through explicit construction and method calls. By contrast, arrays are accessed directly through an indexing operator, the square brackets (”[” “]”). Given this language-level integration, it is necessary to incorporate a visualization for arrays into the contour model in order to satisfy our desire for syntactic completeness. This is an important inclusion in the contour model semantics despite some academic trends to move instruction away from machine-oriented arrays towards abstraction-oriented collections [see *e.g.* 89].

Java’s separation of primitive and reference types complicates the visualization of arrays. In an array of primitives such as an integer array, each cell of the array holds a single integer value. It is natural then to represent the array as a series of boxes, each holding a value; this visualization of arrays is practically as old as arrays themselves, occurring in nearly every computer science textbook that covers arrays. However, when the array contains reference types, this representation is no longer adequate. Each cell of such an array is a reference to an object, not an object itself. The distinction between these two types of Java arrays can be clarified with a comparison to C: the former is like a `T*` array and the latter is like a `T**` array, where `T` is some type.

Arrays of primitives are represented in a contour diagram in the classical manner: with a series of boxes the length of the array, where each box shows the literal for the cell’s value. For arrays of

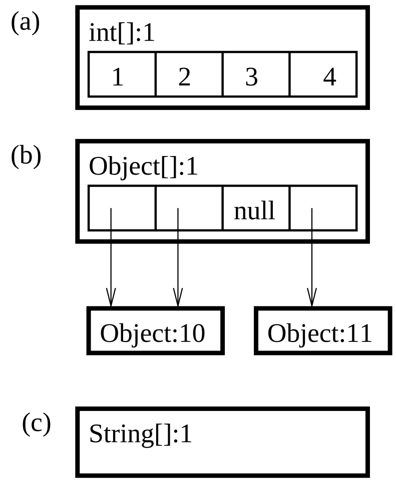


Figure 3.13: Three arrays: (a) an array of integers, (b) an array of `Object`, and (c) an empty array of `String`. Each cell of the integer array contains an integer, but each cell of the object array contains a reference to an object. In this example, the first two cells of array (b) reference the same object.

a reference type, the visualization of arrays as connected, value-holding boxes can still be used, but the values must either be structural links or `null`. In either case, it is also necessary to wrap the boxes within a labeled contour; this maintains consistency in the model and allows for convenient representation of empty arrays (arrays whose size is zero). That is, without a surrounding *array object contour*, there would be no consistent way to represent empty arrays. Array contours are named as any other object contour except that the array annotation "`[]`" is included after the type name and before the instance count. Figure 3.13 demonstrates our representation for arrays. Previous examples, such as Figure 3.11, disguise the `String[]` argument of the `main` method as being `null`. In fact, if a program is started through the `main` method and no arguments are given, then the parameter references an empty array, not null. However, since our examples do not use program arguments, we can represent the parameter as null without loss of generality.

Access Control Modifiers

In Java, access control modifiers can be used to control the scope and visibility of class members. By specifying a member as `private`, `public`, `protected`, or the default package-level access, a programmer specifies how a member can be accessed from the scope of other classes. These modifiers reflect fundamentally design-time concepts, and they are tested and enforced at compile-time. While Java's reflection libraries can be used to process members at run-time, the primary purpose of access control modifiers is to enforce concepts of encapsulation in object-oriented design.

The contour model for object-oriented programs is used to depict and clarify run-time states.

```java
public class HelloWorld {
 private static String message = "Hello, World!";

 public static void main(String[] args) {
  System.out.println(message);
 }
}
```

Listing 3.12: Java HelloWorld program.

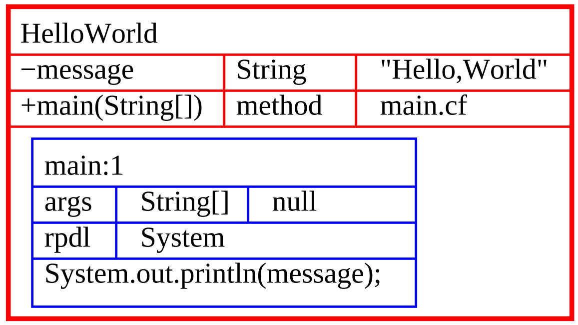


Figure 3.14: An object diagram with access control modifier annotation.

Once compilation has completed, there is almost no need for access control modifiers (modulo reflection), and hence, we rarely need a representation for them. The source code for a program can generally be consulted for static information such as the specification of access control. However, for the sake of completeness, we provide a syntax and semantics for access control modifiers. We adopt the notation of the Unified Modeling Language in another attempt to help close the loop between design specification and program execution. The UML specifies the plus sign (“`+`”), the hash (“`#`”), and the minus sign (“`-`”) to represent public, protected, and private access, respectively. In the same way, the corresponding symbol can be placed within a contour’s member table, preceding each member’s name. Members without access control modifiers are not symbolically annotated in the member table. An example using this notation is provided in Listing 3.12. An object diagram for this example is given in Figure 3.14. Both `message` and `main` have indicators of their access level.

3.3 Representing Java Semantics

The previous section described the syntax used in the contour model for Java. In this section, we justify the representation by illustrating how it clarifies various semantics of object-oriented program execution. Specifically, we examine the behaviors of inheritance, scope, and Java’s oft-confused semantics for its `this` and `super` keywords.

Input: Contour model M, identifier n, referencing context c
Output: Contour whose member table defines n

```
repeat
    if c does not contain a definition for n then
        c ← parent of c;
    end
until n is found;
return c
```

Algorithm 3.1: SimpleLookup: Look up a referenced identifier in the contour model for general object-oriented languages [53]. In a syntactically valid program, this algorithm will always terminate with the correct defining context for n.

Inheritance and Scope

Inheritance is represented in the contour model by nesting subclass contours within their superclass contours. This strategy of nesting applies to static as well as object contours. The nesting of contours works in tandem with the scope resolution rules of the contour model to reflect Java's static scoping of members. In the traditional contour model for object-oriented programs, there is one simple rule for looking up a symbol in the member tables, presented here as Algorithm 3.1. Essentially, contours are searched from deeply-nested to shallowly-nested for the identifier. This technique works perfectly in an idealized statically-scoped object-oriented language, as described by Jayaraman and Baltus [53]. However, this technique cannot be applied directly in Java due to its handling of static members.

It is important to note that the Java language specification does not distinguish between instance and static members for defining scoping rules [41]; this distinction emerges only in the contour model visualization of enhanced object diagrams. Java's scope rules (JLS §6.3) can be interpreted into the contour model according to the following heuristic: given a symbol that is to be resolved, if it is not a local variable, then instance space is searched first, followed by static space, on a contour-by-contour basis. This technique is presented as Algorithm 3.2.

Given an identifier to be resolved, it is imperative that the contextual qualification of the identifier be taken into account. For example, if a method is accessed through an object reference, then the method identifier must be sought in the target object's contour, not within the calling context. In this case, the search begins with the deepest nested object contour of the object's contour group; this reflects the fact that Java's methods are all *pure virtual* under C++ parlance. Figure 3.15 provides an example of this indirection, and Algorithm 3.3 presents an algorithm for resolving these compound identifiers.

Input: Contour model M, identifier n, Referencing contour c
Output: Defining contour of m

repeat
    if n *is defined in* c **then**
        return c
    else
        if c *is an object contour* **then**
            if m *is defined in the static contour for* c **then**
                return *static contour for* c
            end
        end
        $c \leftarrow$ parent of c;
    end
until n *is found*;

Algorithm 3.2: ResolveIdentifier: Resolve an identifier within the contour model. This Java-specific algorithm contrasts with the generic technique presented as Algorithm 3.1.

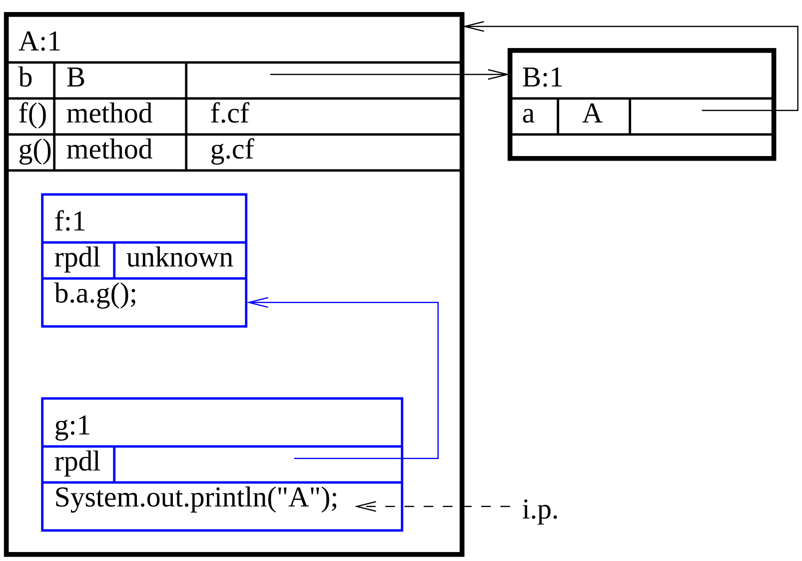


Figure 3.15: Example of member qualification in a method. To resolve `b.a.g()`, one must first follow the object reference for `b`, then the reference in `B:1` to `a`, then the method `g()` can be found in contour `A:1`.

Input: Contour model M, compound identifier $n_1.n_2.\ldots.n_k$, referencing contour c
Output: Defining contour of n_k

foreach n_i, $1 \leq i \leq (k-1)$ **do**
  $c \leftarrow$ ResolveIdentifier(M, n_i, c);
end
if n_k *is a method* **then**
  while *c has a nested object contour o within its object group* **do**
    $c \leftarrow o$;
  end
end
return *ResolveIdentifier(M, n_k, c)*

Algorithm 3.3: ResolveCompoundIdentifier: Resolve the scope of a compound identifier. We assume that identifiers n_1 through n_{k-1} are variables, not methods. Methods require execution and are effectively replaced by a reference to their result in scope resolution.

```
class A {
  int x = 10;
  int f() { return 10; }
  void go() {
    System.out.printf("%d %d", this.x, this.f());
  }
}
class B extends A {
  int x = 20;
  int f() { return 20; }
  public static void main(String[] args) {
    A a = new B();
    a.go();
  }
}
```

Listing 3.13: Java program using `this` and `super`.

This and Super

The keywords `this` and `super` were mentioned in the context of object and static contours in Section 3.2. These keywords provide an explicit method in Java for accessing the current object context or the superclass context, respectively. However, there is a sense in which `this` behaves differently depending on whether a method or a field is being referenced.

Consider the program of Listing 3.13. Figure 3.16 provides an object diagram for the program at the point where the output, "10 20," has just been generated. The expression `this.x` in `go()` refers predictably to its own `x` as in Figure 3.8. However, `this.f()` does not reference `f` in A but rather the implementation in B. The reason for this is that whenever a method is called in Java, the definition used is the one that is the deepest nested in the inheritance hierarchy. In Figure 3.16, this means that although A defines `f`, the implementation in the subclass B is used since it is "deeper"

```
class Queue {

 private List q = new java.util.LinkedList();

 void add(Object elem) {
  q.add(elem);
 }

 Object remove() {
  return q.remove(0);
 }

 int size() {
  return q.size();
 }
}


class MonitoredQueue extends Queue {

 private int maxLength=0;

 void add(Object elem) {
  super.add(elem);
  maxLength=Math.max(maxLength,size());
 }

 Object remove() {
  Object result = super.remove();
  maxLength=Math.max(maxLength,size());
  return result;
 }
}
```

Listing 3.14: Java Queue implementations illustrating how overridden methods can be called through `super`. A contour diagram for this program is shown in Figure 3.17.

in the class hierarchy. This feature of Java is represented in the nesting of method contours `go:1` and `f:1` in the contexts that define them rather than the contexts that use them. Notice that the member table definition for `f` within `A:1` contains two definitions: the local definition as well as the one within the subclass `B`. At the time `B:1` is created, it is known that the method `f` defined in class `A` is overridden, and so this is our visual representation of the overriding of the method. Note that the definition from the class `A` can be accessed either through `super` or the more complex Java notation `A.this.f`.

Java allows overridden methods to be called using the `super` keyword. The rule for nesting of method contours is unchanged in this situation. An example of explicitly calling overridden methods is provided in Listing 3.14. The program provides an implementation for a queue that monitors its

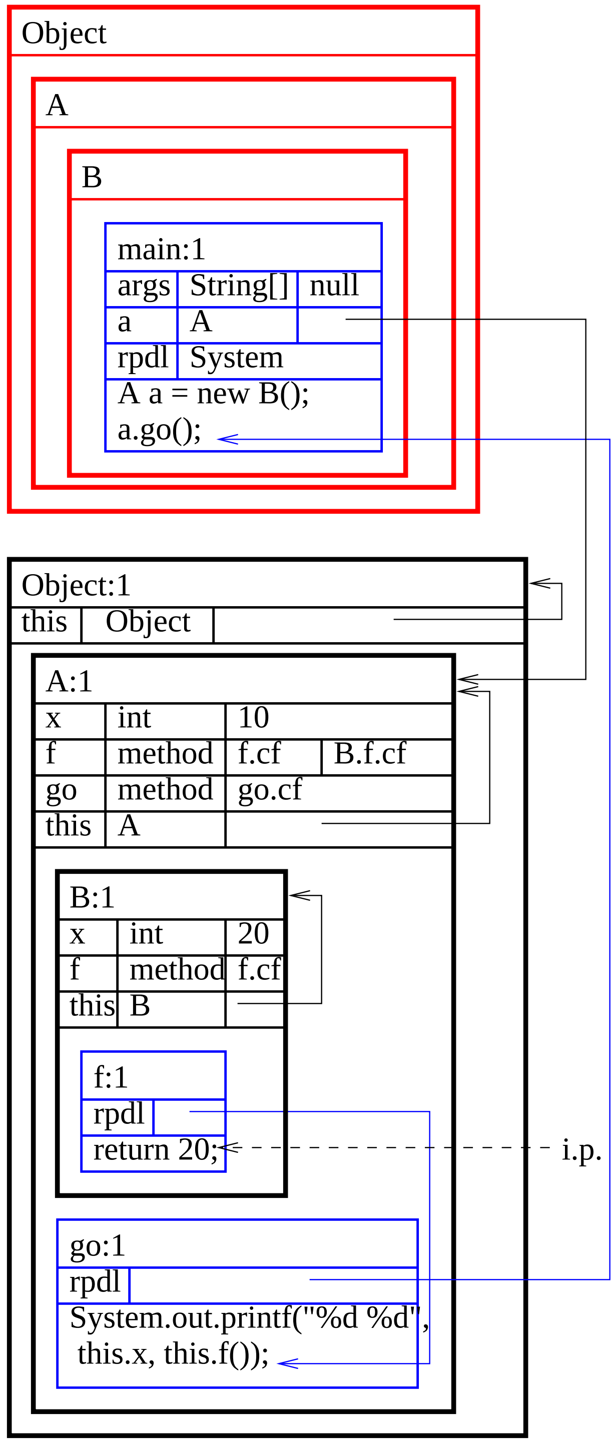


Figure 3.16: Object diagram to clarify how `this` behaves in Java. This program outputs "10 20." In `go:1`, `this.x` is `x` in `A:1`, but `this.f()` calls `f()` in `B:1`.

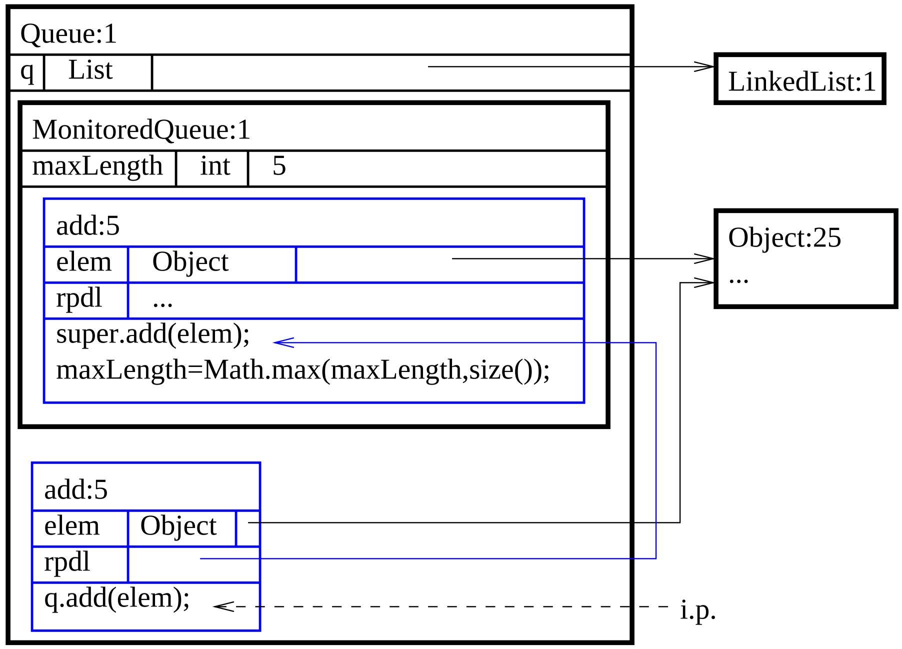


Figure 3.17: Calling an overridden method using `super`. This is an object diagram corresponding to Listing 3.14.

length, recording the maximum length the queue reaches. Figure 3.17 shows how the nesting of method contours reflects the location of the method's definitions, even when overridden methods are called using `super`.[2]

3.4 Multiple Views

The fact that object diagrams can become very large is evident from the previous examples. The greatest advantage of our notation is in its clarity and expressiveness, but this entails requiring a large amount of space to draw the diagrams. but easily its greatest disadvantage is the space that diagrams require. In order to address this problem, we formally introduce some methods for drawing more compact diagrams. When our methodology is used to draw diagrams manually, the composer of the diagram can choose the level of detail to be used. For example, in many of the previous figures we omitted the zero-length `String` array parameter of the `main` method. Interactive tools cannot benefit from this subjectivity, and so JIVE requires more formal models for multiple views. A user should be able to select from a variety of visual paradigms so that the diagrams are suited to his or her needs. Accordingly, support for multiple views at different levels of granularity is a fundamental requirement for interactive visualization of object-oriented programs, as described in

[2]One could argue that better implementations would use delegation or an observer, but the given program suffices as an example. It is also the type of design one might find produced by a student in an introductory course.

the introduction.

There are essentially two ways to save space in a contour: change the amount of detail shown within it or change which portions that are visible. First, we will consider the member table, which can be very large for classes or methods of moderate complexity. Within the member table, we have found that the entries for method definitions are rarely used beyond explaining the basic semantics of overriding. Hence, these entries are often omitted, as was demonstrated the `Queue:1` and `MonitorQueue:1` contours of Figure 3.17. Similarly, the entries for `this` and `super` can be omitted unless the intention is to communicate explicitly how these keywords can be used to connect subclasses and superclasses (*e.g.* Figure 3.15). It is also useful to filter method entries by access control modifier; hiding `private` members makes sense when one is not interested in a particular object's implementation, for example.

The four basic sections of the contour are its title (*i.e.* the contour's name), the member table, the area for nested contours, and the source code attachment, and any of these can be hidden in order to save space or hide detail. Figure 3.10, for example, presented an example in which the member tables have been hidden in all the object contours for the class `Object`. In object contours, we tend have consistently omitted the source code section. It is often redundant to show both the source code and the member table since the member table is derived from the source code. Within method contours, displaying source code section has more merit, but it can still be hidden, especially if the source is viewable outside of the object diagram entirely. JIVE provides the capability to have a separate source code viewer that highlights program execution [38]. Finally, the nested contours area can be hidden if such details are not important; to focus on the object structure, one might wish to hide method activations.

Certain types of views have proven to be useful in the development and testing of JIVE. We have defined the following paradigms for drawing contours:

Minimized A *minimized contour* is one that has been reduced to a single point. The point is optionally labeled with the name of the contour. For minimized object contours, the name of the deepest nested non-inner object contour is used; this is the most specific class of the object. For static contours, the outermost contour name is used; this policy labels the point as the most generic class, which must by definition contain the static contours for all loaded subclasses. Structural and return links are unaltered by minimization.

Collapsed A *collapsed contour* has had its superclasses hidden, but it is otherwise shown normally. Several of our examples have used this view, such as Figure 3.15, in which `A:1` and `B:1`

```
class A {
 void f() {
  System.out.print("Hello, ");
 }
}

class B extends A {
 void f() {
  super.f();
   System.out.println("World!");
  }
}
```

Listing 3.15: A Java program illustrating the complications of showing superclass method activations in collapsed view. A contour diagram for this program is shown in Figure 3.18.

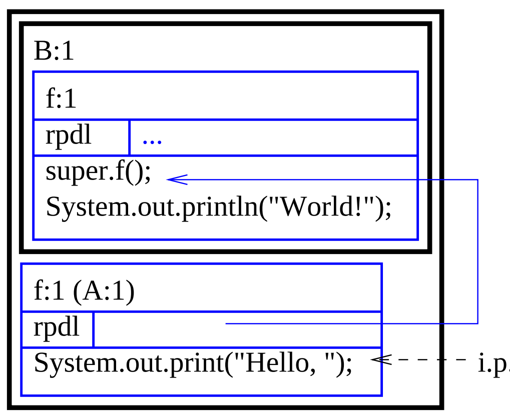


Figure 3.18: Calling a method in a hidden superclass of a stacked object contour. This contour diagram represents a state of the program shown in Listing 3.15.

are collapsed so that their parent `Object` contours are not shown. It is possible to stack a method contour, which would hide not only its containing contour's superclass contours but its containing contour itself. However, this is generally a bad practice since methods should be shown within object contexts according to our desiderata for visualization.

A fundamental complication of the collapsed view is that there is a visual incongruity when it comes to showing methods that should be nested within a hidden containing contour. The simple program of Listing 3.15 exemplifies the problem with a case a hidden contour's method is called: Figure 3.18 illustrates our solution, which is to expand the collapsed contour so that only the nesting area of its appropriate superclass is shown, and then to relabel the method contour with the name of its context.

Detailed When a contour is shown with all of its sections visible, even if some (but not all) elements of the member table are hidden, the contour is in *detailed view*.

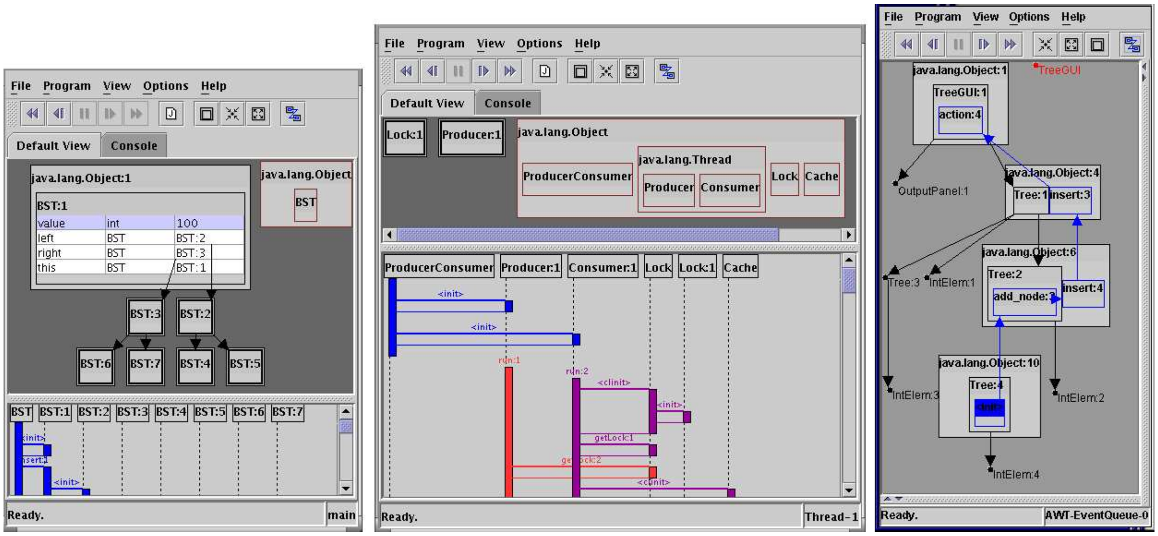


Figure 3.19: Three screenshots of JIVE. The left and center shots show object diagrams in their top halves, with stacked-view objects shown in black and static space in red. These two screen shots also show how JIVE integrates a view of state (the object diagram) with a view of execution history (the sequence diagram, in the bottom portion of the screenshots). The right screenshot shows a call-path diagram, where only objects with active methods are shown.

The previous visual paradigms apply to individual contours. Object diagrams themselves may contain contours that are all viewed in the same paradigm, or they may contain contours viewed in many different styles. A *minimized diagram* has all of the contours in minimized view. A *detailed diagram* has all contours in detailed view. A *call-path diagram* shows objects with active methods in detailed or stacked view, while objects without methods are minimized. Figure 3.19 shows three snapshots of JIVE that illustrate its support for these various types of views. This is not an exhaustive enumeration of combinations of views; rather, these are some of the views we have found to be useful in practice.

3.5 Mathematical Representation

It is convenient to have a formal mathematical model for contours so that operations on the contour model can be precisely defined. A contour model can be expressed as a set of contours and the relationships among them. The three types of contours, in order of increasing complexity, are static, object, and method, and we present definitions for each respectively. We define the *class* function as a convenience to presenting the mathematical representation of the contour model: given an object or static contour v, $class(v)$ returns the Java class that corresponds to v. That is, given a static contour v for a class `C`, $class(v)$ returns `C`, and for an object contour $v : n$, $class(v : n)$ also returns

`C`.

A static contour can be defined as a tuple $v = \langle S, N, A, M, E \rangle$, where:

- S is the set of static contours nested immediately within v.
- N is the set of object contours immediately nested within v.
- A is the set of object contours referenced by v through a structural link. That is, for each $a \in A$, there is a field in the member table of v whose value is a reference to a.
- M is the set of method contours nested immediately within v.
- E is the environment of v. This is a mapping from identifiers of v to their denotations. Therefore, E is the symbolic representation of the member table of v. Note that $A \subseteq E$, since A is determined by those elements of E that reference objects.

An object contour is similar in structure to a static contour. Each object contour must have a link to its static counterpart in the contour model even though this static link is generally not shown in a contour diagram. An object contour then can be defined as $v = \langle s, N, A, M, E \rangle$, where:

- s is the static contour counterpart to v.
- N is the set of object contours immediately nested within v.
- A is the set of object contours referenced by v through a structural link.
- M is the set of (non-static) method contours nested immediately within v.
- E is the environment of v. This includes the values for virtual members such as `this`. As with static contours, $A \subseteq E$ holds.

Method contours represent activations of static and non-static of methods. This is a syntactic distinction in the Java language, but in the contour model semantics, the staticity of a method is reflected in its placement: static methods are placed in static space, and non-static methods are placed either in object space or within an inner object contour in static space. A method contour can be defined as $m = \langle A, E, r, t, I, ip \rangle$, where:

- A is the set of object contours referenced by v through a structural link.
- E is the environment of m, where $A \subseteq E$.

	May contain	**May reference**
Object contour	object, method, and instance inner object contours	object contours, static contours through static link only
Static contour	static, method, static inner object, and instance inner object contours	object contours
Method contour	none	object contours, method contours through return links only

Table 3.1: Legal nestings and connections in a contour model

- r is the return contour of m. That is, r is the method contour that called m. We assert that $r \in M \cup \{\lambda\}$, where M is the set of method contours and λ is represents the fact that a method was called directly by the Java Virtual Machine. This is the case, for example, when m is the first method on its thread.

- t identifies the thread on which the method is executing.

- I is the instruction sequence of m.

- ip is the instruction pointer indicating where control resides in the method. This can be a null value if the method does not currently have execution focus, such as if it has called another method that is currently executing.

The last element, I, is an optional component of the tuple. There are cases where the instruction sequence may not be supplied, for example, in a library method.

A contour model can be defined as $M = \langle s, O\rangle$, where s is the root static contour and O is the set of object contours that comprise object space. In Java, static space is always rooted in a single static contour for `java.lang.Object`, which implies that either s is a static contour for `Object` or the contour model is empty. Any contour in the contour model can be reached through its enclosing contour, and so the $M = \langle s, O\rangle$ tuple definition is sufficient to represent all contours. The connections among contours through structural and return links are encoded within the contours themselves. Table 3.1 summarizes the syntactic legality of contour nesting and connection through structural and return links.

Given a graph $G = \langle V, E\rangle$, it is common to refer to the V and E components of G as $V(G)$ and $E(G)$, respectively. This notation is also applicable to the contour model definition. For example, given an object contour $v = \langle S, N, A, M, E\rangle$, we use the notation $N(v)$ to reference the set of nested object contours (N) of v. This notation facilitates the discussion of contour models and the specification of algorithms over them. Additionally, it is convenient to use the notation $v \in M$ to mean that a contour v is in the contour model M, although the formal specification of this notation is more complicated.

It is useful to define three functions *name*, *type*, and *value* that return the name, type, and value of any element in a contour's member table. Object, static, and method contours define an environment E that maps identifiers within the contour to their denotations. These functions are informal and used to facilitate discussion about the contents of a contour's member table. As an example, a method activation may contain a field `i` of type `int` whose value is 15. Given a method contour m representing this activation such that $E(m)$ contains a field f corresponding to the aforementioned variable `i`, then the following are true: $name(f) = \texttt{i}$; $type(f) = \texttt{int}$; and $value(f) = 15$.

Referential Integrity in the Contour Model

The presentation of contour model semantics thus far has assumed that complete program information present; that is, we have assumed that all details of a program's execution are known and interpreted in the contour model. In practice, there are cases where complete information is not desired or, as explained in Chapter 5, impossible to obtain. For example, given logging capabilities such as the JPDA-based one described in Chapter 5, information on native function calls cannot be gathered. It is impossible to generally convert native libraries into contours within the Java contour model for two reasons: first, native libraries can be written in any language and do not necessarily follow the semantics we have presented; and second, the JPDA-based application architecture described in Chapter 5 does not allow access to native libraries. Large applications provide a clear example of cases where it is undesirable to record complete information. Often there are many operations that are of no interest to the user of the visualization system, and so it would be wasteful, both in computational time and storage space, to build an complete record of program execution. Hence, it is often the case that only a partial contour model is recorded and processed.

We define a *complete* contour model as a contour model M that exhibits the following properties:

Property 3.1: Thread completeness. For all method contours $m \in M$, either $r(M) \in M$ or $r(M) = \lambda$. That is, for all methods, the return contour is either in the model or the method is the first on its thread.

Property 3.2: Referential completeness. For all fields f in any contour in M, if $type(f)$ is a reference type and $value(f)$ is an object contour v, then $v \in M$. That is, the objects referenced by all fields are in the contour model.

Property 3.3: Parent completeness. For all contours $v \in M$ such that v is nested in contour p, $p \in M$. That is, the parent of all contours is in the contour model.

Property 3.4: Static completeness. For all object contours $v \in M$, if *type*(v) is not an instance inner class, then $s(v) \in M$. That is, there is a static counterpart in the contour model for every object contour that should have one.

A contour model that satisfies all of the above properties is *complete* and *well-formed*. If a contour model does not satisfy all of Properties 3.5 through 3.5, it is an *incomplete* contour model. An incomplete contour model may still be well-formed as long as the following properties hold for a contour model M:

Property 3.5: Enclosing context completeness. For all inner objects $v \in M$, where the enclosing contour of v is p, then $p \in M$. That is, inner objects cannot exist in the contour model without their enclosing contours.

Property 3.6: Object replacement. Given a field f contained with in a contour $v \in M$, if f refers to a contour $t \notin M$, then $value(f)$ is specified with a string identifier; furthermore, for all fields $g \in M$, $value(f) = value(g)$ if and only if f and g reference the same object.

Property 3.7: Return link replacement. If a return link r of a method contour $m \in M$ references a method contour $n \notin M$, then r is specified as a unique string identifier. Furthermore, if there is a sequence of method contours $m_1, m_2, \ldots, m_k$ such that the following all hold:

- there is a thread identifier t such that $t(m_i) = t$ for all $i \leq i \leq k$;
- $m_1 \in M$;
- $m_k \in M$;
- m_i calls m_{i+1} for all $1 \leq i \leq k-1$; and
- there is an m_j $(2 \leq j \leq k-1)$ such that $m_j \notin M$;

then we assert that $r_1(r_2(\ldots r_{k-1}(m_k)\ldots)) = m_1$, where $r_i(v) = r(v)$ (*i.e.*, the subscript i of r_i is simply for numbering). That is, if there is a sequence of method calls $m_1, m_2, \ldots, m_k$, and there is at least one method in m_2 through m_{k-1} that is not in the model, then we can still reach m_1 from m_k by following k return links.

The definition of incomplete contours is a necessity for practical applications of the contour model semantics within the JIVE methodology. However, unless explicitly specified, we restrict the discussion to complete contour models, which allows a clearer explanation of algorithms and properties. For the most part, dealing with incomplete contour models is a matter of implementation details and creating a visualization architecture that supports the incomplete information contained in the model.

Chapter 4

Understanding Program Execution using JIVE

In this chapter, we present four examples of how JIVE can be used to understand program execution. The JIVE system's advanced capabilities present a novel and unique approach to facilitating program understanding. The combination of state and history visualizations, along with interactive execution, provides the user with flexible control over how they wish to interact with a program at runtime. The four examples we will show in this chapter illustrate the broad applications of JIVE:

- Section 4.1 presents a student's experience using JIVE's enhanced object diagrams to debug a class project.
- Section 4.2 describes how enhanced object diagrams can clarify Java's overriding and shadowing semantics for members.
- Section 4.3 demonstrates how JIVE can be used to understand the *visitor* design pattern, one of the most complicated patterns described by Gamma et al. [32].
- Section 4.4 presents an example of how a student used JIVE's automatic sequence diagrams to debug an application reifying the *iterator* design pattern.

4.1 Clarifying Object Structure

In order to demonstrate the effectiveness of JIVE's enhanced object diagrams, we present an example of JIVE was used to debug a class project. The student's program was written as a requirement for a graduate class in programming languages, and the task was to simulate inheritance by delegation in the context of a binary search tree. The students were given a set of binary search tree classes that were related through generalization, using the architecture presented in Figure 4.1. This program

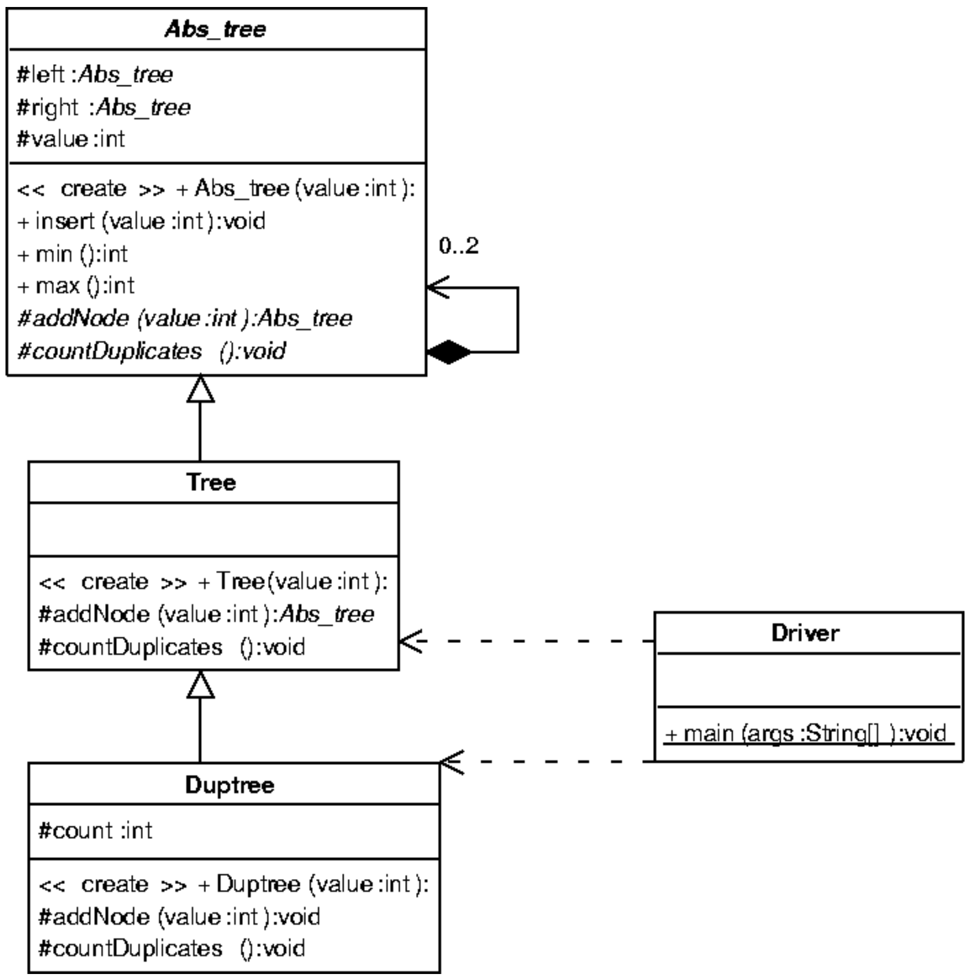


Figure 4.1: A class diagram of a set of binary search tree classes designed using inheritance. `Abs_tree` is an abstract tree that contains the basic elements of a tree. `Tree` is a simple BST implementation that ignores duplicates. `Duptree` extends `Tree` to count duplicates.

```
public class Driver {
    public static void main(String args[]) {
        int i;
        Tree tree = new Tree(100);
        IAbs_tree absTr = tree;
        ITree tr = tree;

        absTr.insert(tr, 50);
        absTr.insert(tr, 150);
        absTr.insert(tr, 75);
        absTr.insert(tr, 175);
        absTr.insert(tr, 25);
        absTr.insert(tr, 125);

        Duptree duptree = new Duptree(100);

        absTr.insert(tr, 50);
        absTr.insert(tr, 150);
        absTr.insert(tr, 75);
        absTr.insert(tr, 175);
        absTr.insert(tr, 25);
        absTr.insert(tr, 125);
    }
}
```

Listing 4.1: Erroneous `Driver` implementation

defines an abstract `Abs_Tree` class that contains the basic elements of a binary search tree: a comparable value (an integer in this case) and left and right fields.

The student's initial attempt at converting this program to use delegation instead of inheritance yielded the architecture shown in Figure 4.2. We will not address here the correctness of the solution, only the process by which the student debugged the program. The student's solution uses interfaces to define common behavior, and the concrete classes reference the interfaces through fields. Delegation is performed through the interfaces, and so the spirit of the solution is appropriate to the problem. (A superior solution to the problem would take a more general approach: use interface inheritance in place of implementation inheritance and a more mechanical (*i.e.* automatable) source transformation.)

The error was not introduced in the conversion from inheritance to delegation, which was indeed crux of the problem. Instead, the error arose in the student's `Driver` class, within the testing unit. Listing 4.1 illustrates the original and incorrect `Driver` implementation. It appears as though the student has copied the testing case for `absTr` with the intention of using the same code to test `duptree`. However, he has neglected to rename the variable `absTr` to `duptree`, and hence we suspect there will be a problem. This is an instance of a *blunder variable*, to use Donald Knuth's

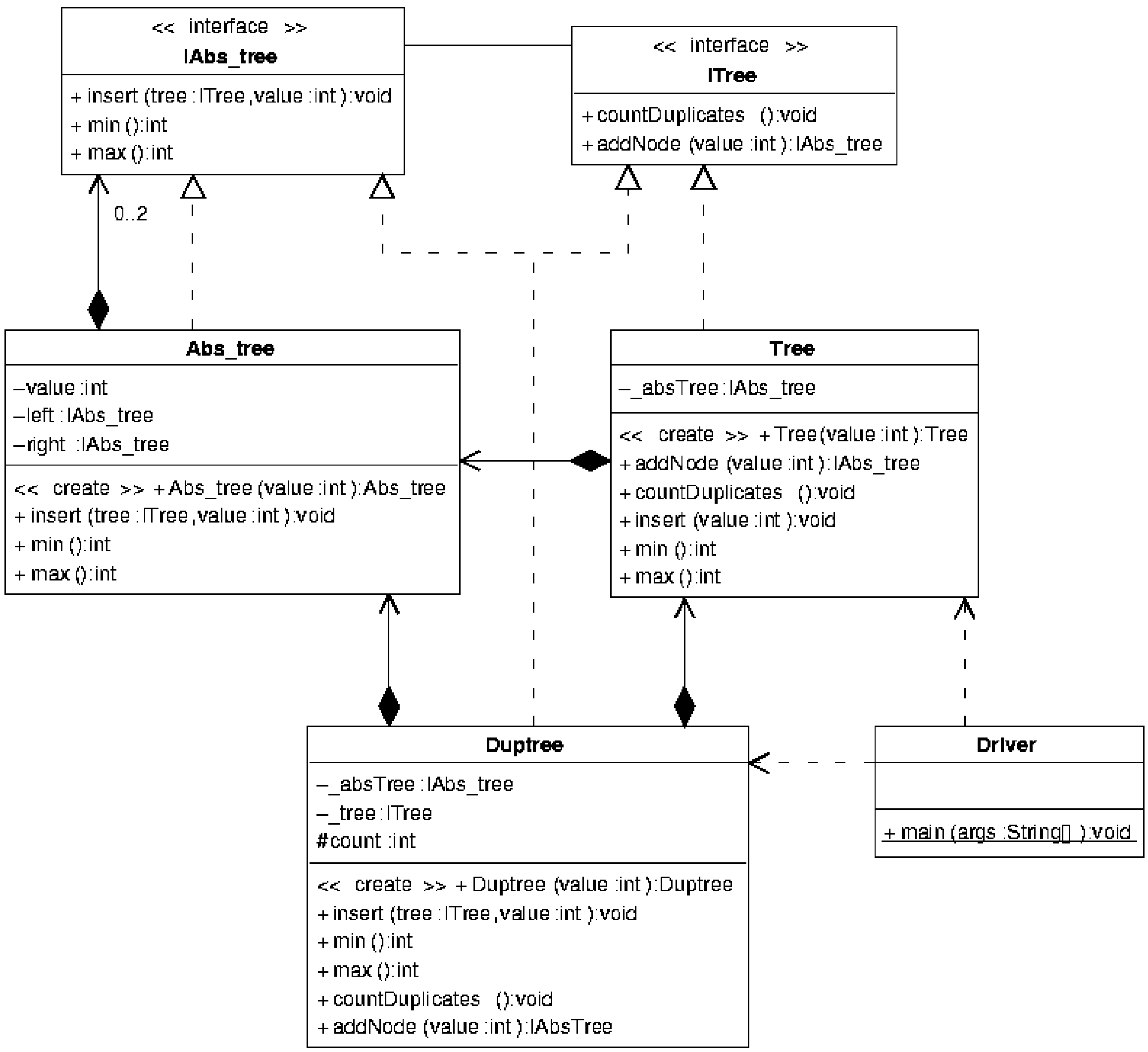


Figure 4.2: A class diagram of a set of binary search tree classes designed using delegation in place of implementation inheritance. This architecture is a modification of Figure 4.1. `Tree`, which previously extended `Abs_tree`, now aggregates it. Similarly, `Duptree`, which previously extended `Tree`, now aggregates both `Tree` and `Abs_tree`.

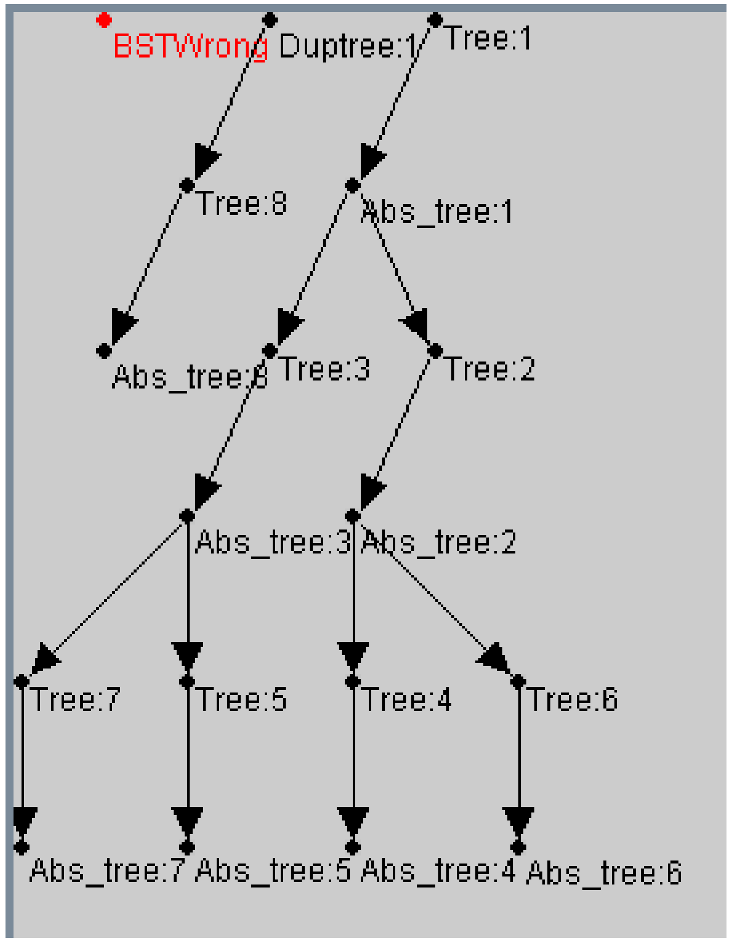


Figure 4.3: A minimized enhanced object diagram generated from the program of Figure 4.2 using the `main` method of Listing 4.1. This diagram indicates that there is a problem with the program since the `Duptree`-rooted structure is too small.

bug ontology [60]. In this case, the student ran the program through JIVE, which produced the object diagram shown in Figure 4.3. Inspecting the minimized enhanced object diagram, the student observes that the tree on the right seems to be properly structured, with each `Tree` object aggregating and delegating to a `Abs_tree` object. However, the tree on the right, rooted in `Duptree:1`, is certainly too small to be correct. Evaluating the `main` method, the student realizes that the `Duptree` is not being properly built at all. He therefore modifies the `main` method to match the one shown in Listing 4.2. Rather than renaming `absTr` in the second testing case, the student has simply reassigned the variable to reference the `Duptree` object. Running the program through JIVE a second time reveals the enhanced object diagram shown in Figure 4.4. In this object diagram, the relative positions of the two trees have changed. The `Duptree` structure is much larger than the other, which is to be expected due to the additional level of indirection (and therefore aggregation)

```
public class BSTequiv {
    public static void main(String args[]) {
        int i;
        Tree tree = new Tree(100);
        IAbs_tree absTr = tree;
        ITree tr = tree;

        absTr.insert(tr, 50);
        absTr.insert(tr, 150);
        absTr.insert(tr, 75);
        absTr.insert(tr, 175);
        absTr.insert(tr, 25);
        absTr.insert(tr, 125);

        Duptree duptree = new Duptree(100);
        absTr = duptree;
        tr = duptree;

        absTr.insert(tr, 50);
        absTr.insert(tr, 150);
        absTr.insert(tr, 75);
        absTr.insert(tr, 175);
        absTr.insert(tr, 25);
        absTr.insert(tr, 125);
    }
}
```

Listing 4.2: Corrected `Driver` implementation

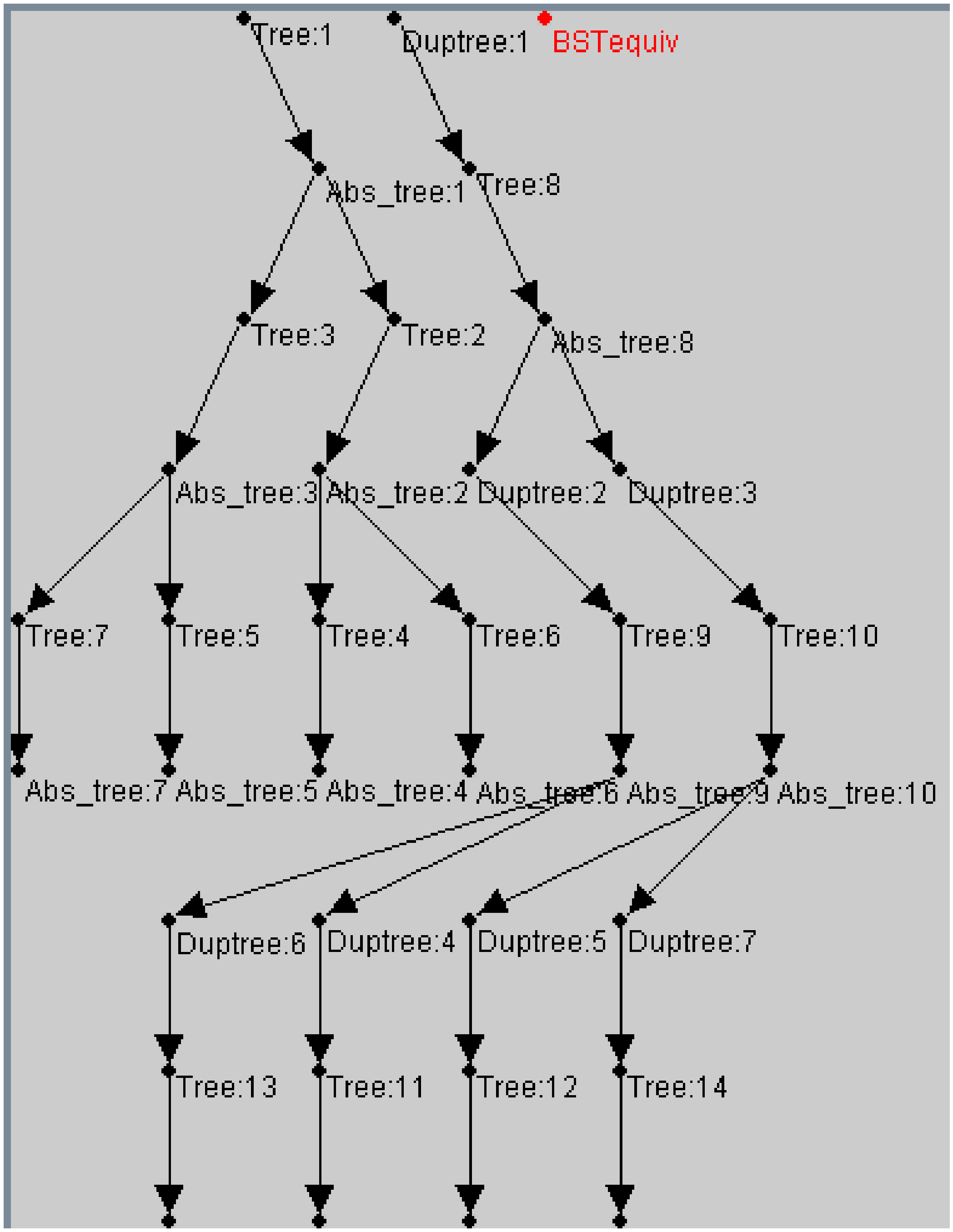


Figure 4.4: A minimized enhanced object diagram generated from the program of Figure 4.2 using the `main` method of Listing 4.2. Compared to the object diagram of Figure 4.3, the structure of the `Duptree`-rooted tree is closer to the intention of the programmer, and hence this diagram illustrates that a bug has been fixed.

```java
class A {
  public int f() { return 500; }
  public int g() { return i+f(); }
  private int i = 1;
}
class B extends A {
  public int f() { return i*i + super.f(); }
  private int i = 2;
}
class Test {
  public void main() {
    A a;
    B b;
    b = new B();
    a = b;
    System.out.println(a.g());
  }
}
```

Listing 4.3: Java program to illustrate overriding vs. shadowing

used in its implementation. The student is now satisfied that the object structure can be considered correct.

4.2 Understanding Java's Scope Rules

In a non-object-oriented, statically-scoped program, the meaning of any identifier is easily resolved. Johnston's classic contour model can be used for such programs without significant modification. However, handling object-oriented variable scoping is complicated by inheritance relationships, as described by Jayaraman and Baltus [53] and presented briefly in Chapter 2. Extending the contour model semantics to Java introduces more complexity to identifier resolution, but Java's static scoping rules can still be properly incorporated, as presented in Chapter 3. The previous examples in this chapter have illustrated how JIVE can be used to debug, to teach, and generally to understand program execution. Our final example will explore more intricate details of the Java language, specifically focusing on how the contour model semantics for Java can be used to clarify program execution when there is identifier overriding and shadowing.

Listing 4.3 presents a program that has been used frequently at the University at Buffalo to teach the difference between overriding and shadowing in Java. Students' confusion in understanding this program's meaning stems from the fact that both `f` and `i` are defined in `A` and its subclass `B`. In Java, methods are strictly *overridden*, and so a method identifier refers to the definition in the most specific implementation (*i.e.* the deepest subclass). However, other implementations may be accessed through the `super` keyword. Hence, given an instance of `A`, the call to `f` in `g` will activate the `f` defined in `A`, but given an instance of `B`, the call to `f` in `g` will activate the `f` defined in `B`. Variables, on the other hand, are **`hidden`**, so that the variables defined in a class hide the definitions in any other class related through inheritance. Therefore, given an instance of `A`, the `i` in `g` refers to the `i` in `A`, and given an instance of `B`, the `i` in `g` refers to the one in `A` while the `i` in `B`'s `f` refers to the `i` in `B`. This wordy description of the program's behavior is accurate, but it is not intuitive, and hence we run this same program through JIVE.

Figures 4.5 and 4.6 show four screenshots of JIVE. The first screenshot shows the basic structure after `a` and `b` have been created. Notice that both `a` and `b` reference `B:1` despite the fact that `a` is declared as type `A`. The second screenshot shows the calling of `g`. The method contour `g()` appears in the context of `A:1`, the contour in which it is defined. The `i` in `g` is therefore correctly determined to refer to `i` in `A:1`, with value 1. The partial evaluation of the expression is currently 1 plus the result of calling `f`. This call to `f` is shown in the third screenshot; due to Java's overriding of methods, the `f` in `B:1` is called, and the method contour `f()` is placed in that contour. The `i` references in this method are correctly determined to refer to `i` in `B:1`, whose value is 2, and therefore the expression $\mathtt{i} * \mathtt{i}$ evaluates to 4. The call `super.f()` invokes explicitly the definition of `f` in `A:1`, and this call is shown in the final screenshot. This method returns 500, and hence the final result that is printed by the program is 505. The contour model semantics therefore follows Java's scope rules and, as an abstract machine, computes the correct value for the application. Once a user has mastered the few basic rules of Java's contour model semantics, then the combination of enhanced object diagrams and interactive execution can clarify program execution beyond plain text or *ad hoc* drawings.

4.3 Understanding the Visitor Design Pattern

The seminal Gang-of-Four book [32] on design patterns consolidated and categorized the common well-structured uses of the principles of object-orientation. Since its publication in 1995, formal design pattern methodology and nomenclature has worked its way into an increasing number of applications, including computer science instruction and industrial training courses. The patterns themselves are described in terms of class structure, object structure, and behavior. However, any printed representation of design patterns has an inherent problem that the depictions of dynamic behavior are themselves static. That is, a diagram representing a dynamic system does not change, and so time-varying behavior must be inferred from notations that are easily misunderstood by novices. More recent publications, such as Allen Holub's book on learning design patterns by

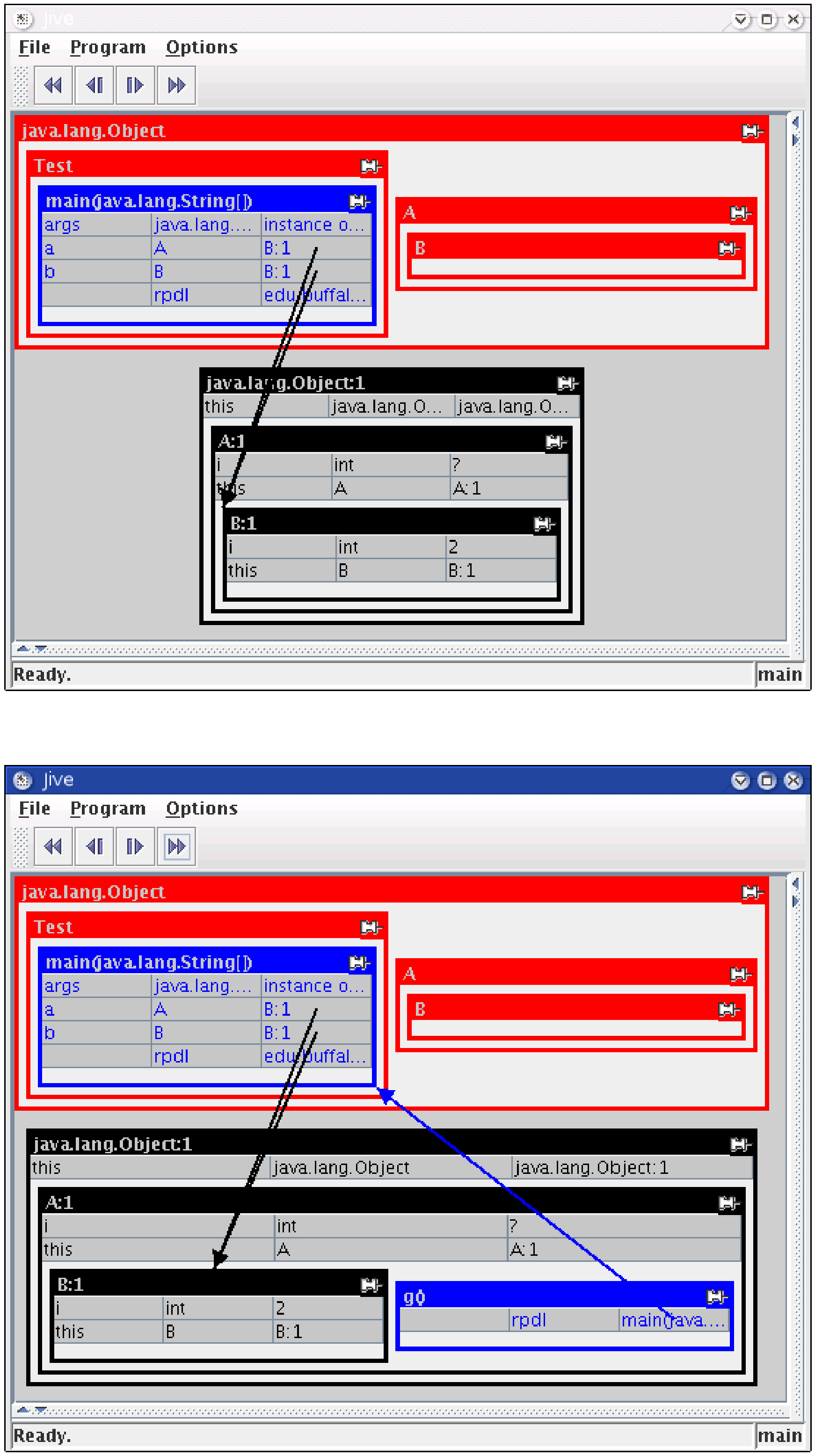


Figure 4.5: First two significant steps of the program of Listing 4.3

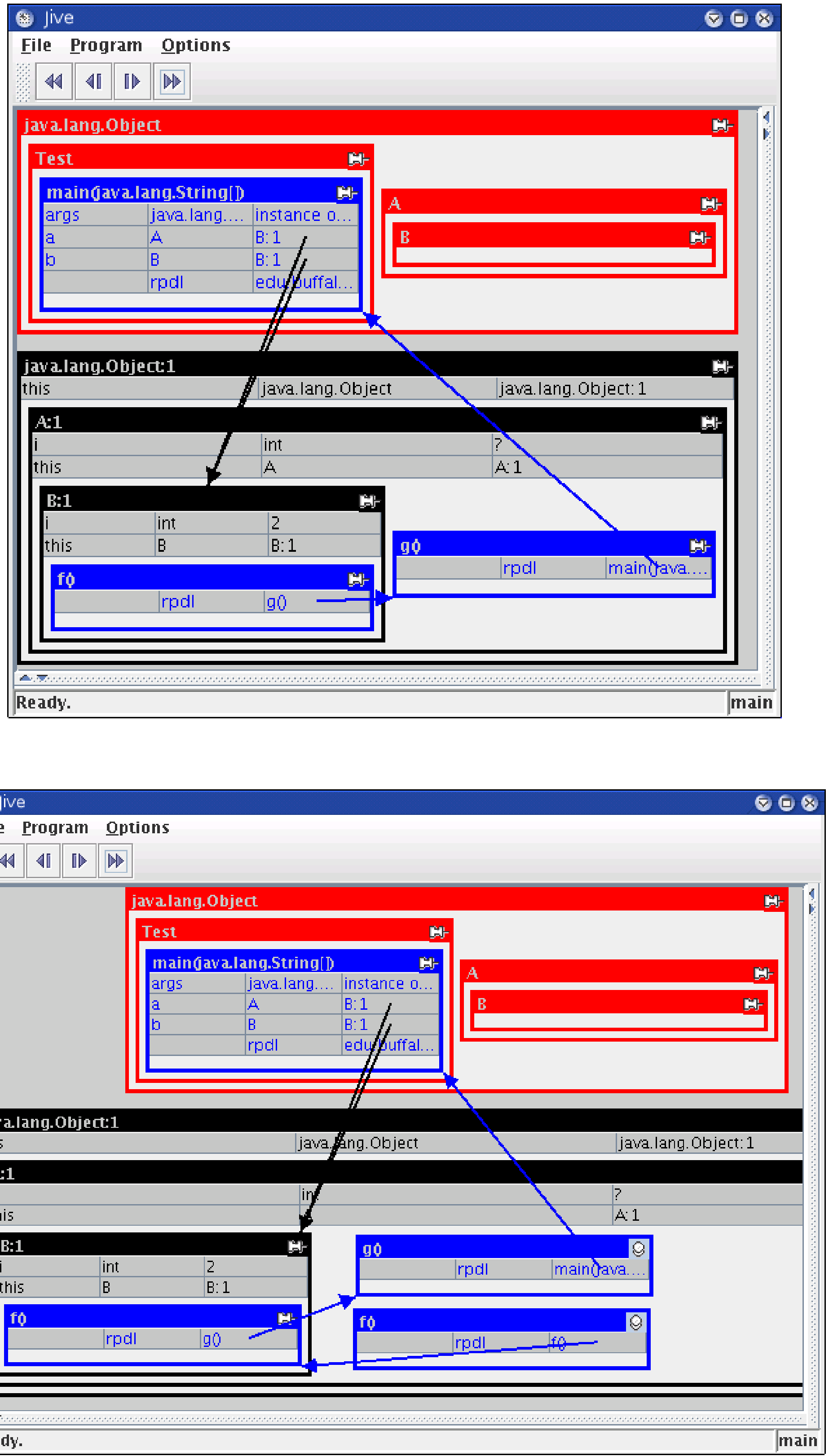


Figure 4.6: Last two significant steps of the program of Listing 4.3

looking at complete code examples [49], wrap the discussion of design patterns in new contexts and languages compared to their original publication, but the difficulty of presenting dynamic systems is still present.

JIVE provides a convenient framework for exploring and understanding the dynamic behavior of systems in general, and so it is natural to extend this pedagogic orientation to design patterns. As an example, we will consider the *visitor* design pattern as described by Gamma et al. [32]. This pattern is founded on *double-dispatch*, the practice of calling methods in different object contexts for an added level of indirection. For reference, the class structure and ideal interactions of the visitor pattern are shown in Figure 4.7. Visitor is considered to be one of the most general patterns, as many other patterns can be implemented using the visitor pattern. Anecdotally, it is also often difficult for students to understand based on the textbook description.

As a reification of the visitor, we will use a linked list whose structure contains two distinct types of nodes: *data nodes*, which contain a reference to an `Object` data element, and *empty nodes*, which contain no additional data. We wish to be able to traverse the list and perform different operations on each node depending on whether it is a data node or an empty node. This is a classic case where the visitor pattern can be applied, and a solution strategy is shown in the class diagrams of Figure 4.8.

A seasoned software developer can look at the visitor pattern, its behavior, and the linked list reification of the pattern, and the dynamic behavior of the system will be clear. However, to a novice, it can be difficult to work at the meta-programming level of design patterns, and frequently students become confused trying to map from the actual program architecture to the stereotypical behavior presented in Figure 4.7. This is where JIVE's dynamically generated sequence diagrams can close the loop, helping a student make the connection between their code and the pattern. Listing 4.4 shows a sample `Driver` implementation for this example, and Figure 4.9 shows a JIVE-generated sequence diagram corresponding to the program's execution. This figure illustrates how JIVE's sequence diagrams present the actual program execution, not idealized representation of the execution, and how this clarifies the relationship between the linked list program and the visitor design pattern. The student can place the JIVE-generated sequence diagram alongside the textbook description of the visitor pattern, allowing him or her to associate `Driver` with *Client* or `StringBuildingVisitor` with *ConcreteVisitor1*. In this way, JIVE can be used as a pedagogic tool to explain design patterns, an advanced topic of computer science, to even introductory level students.

Figures 4.7 and 4.9 also highlight other interesting properties of sequence diagrams. Figure 4.7 exhibits one of two popular paradigms, which we call the *focus-of-control paradigm*. In this paradigm,

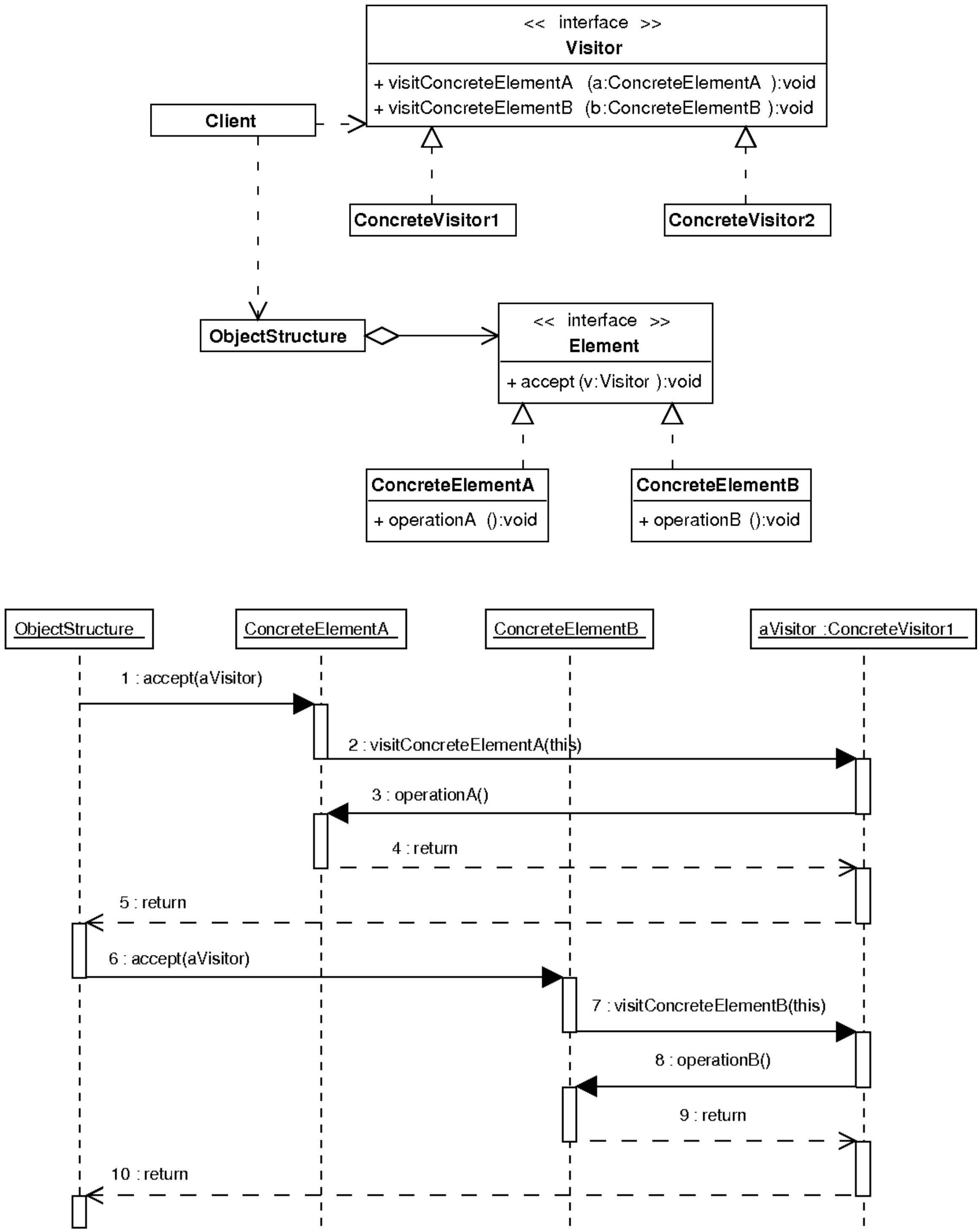
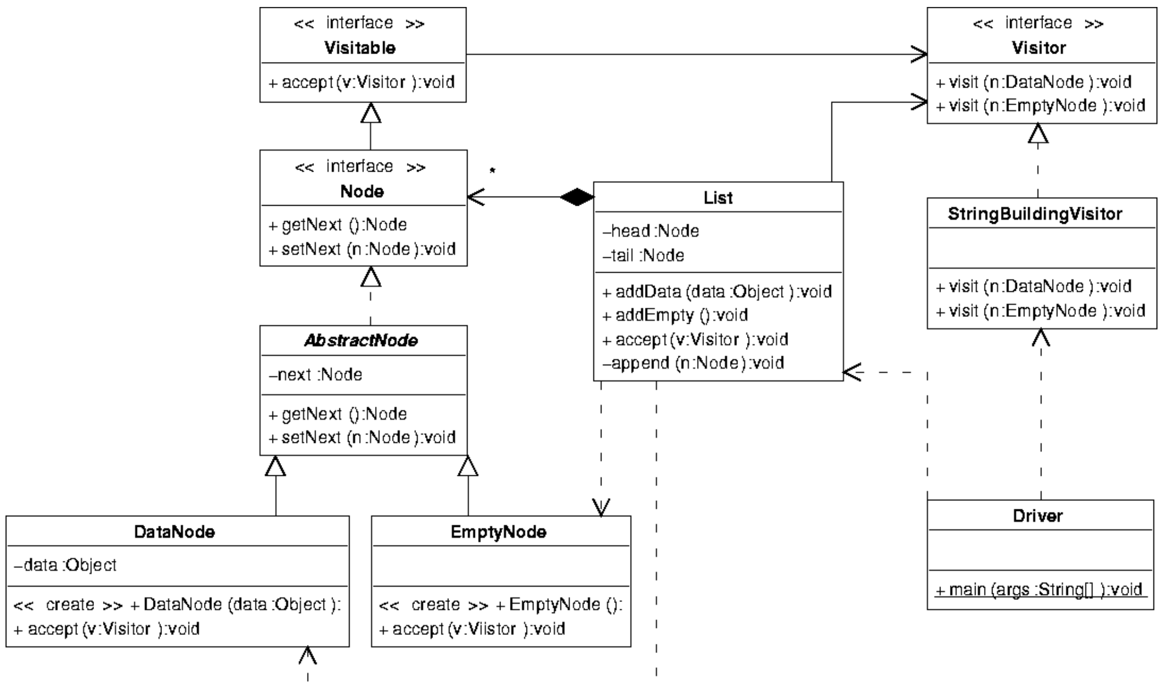


Figure 4.7: The class architecture and stereotypical interactions of the visitor pattern. This figure is adapted from the presentation in Gamma et al. [32]



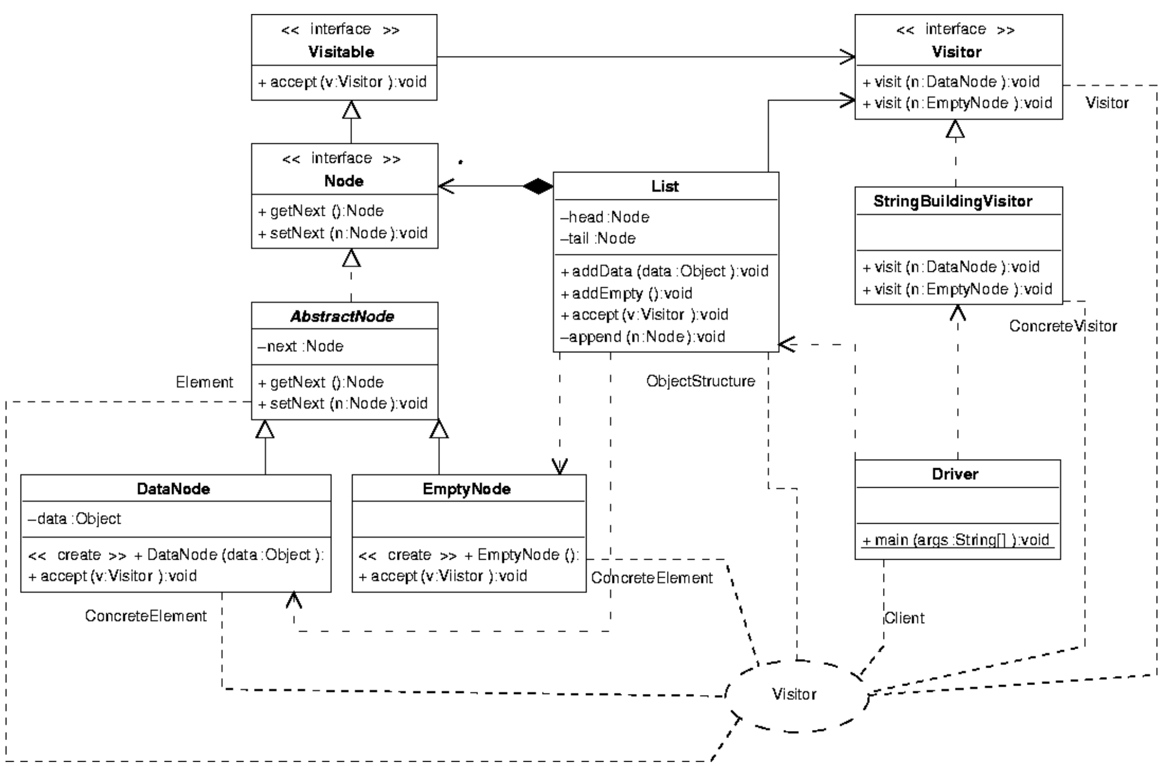


Figure 4.8: Two class diagrams of a linked list application that uses the visitor pattern to process the list's nodes. The bottom class diagram presents the same architecture as the top one, except that the participants of the visitor pattern have been explicitly annotated using a UML collaboration.

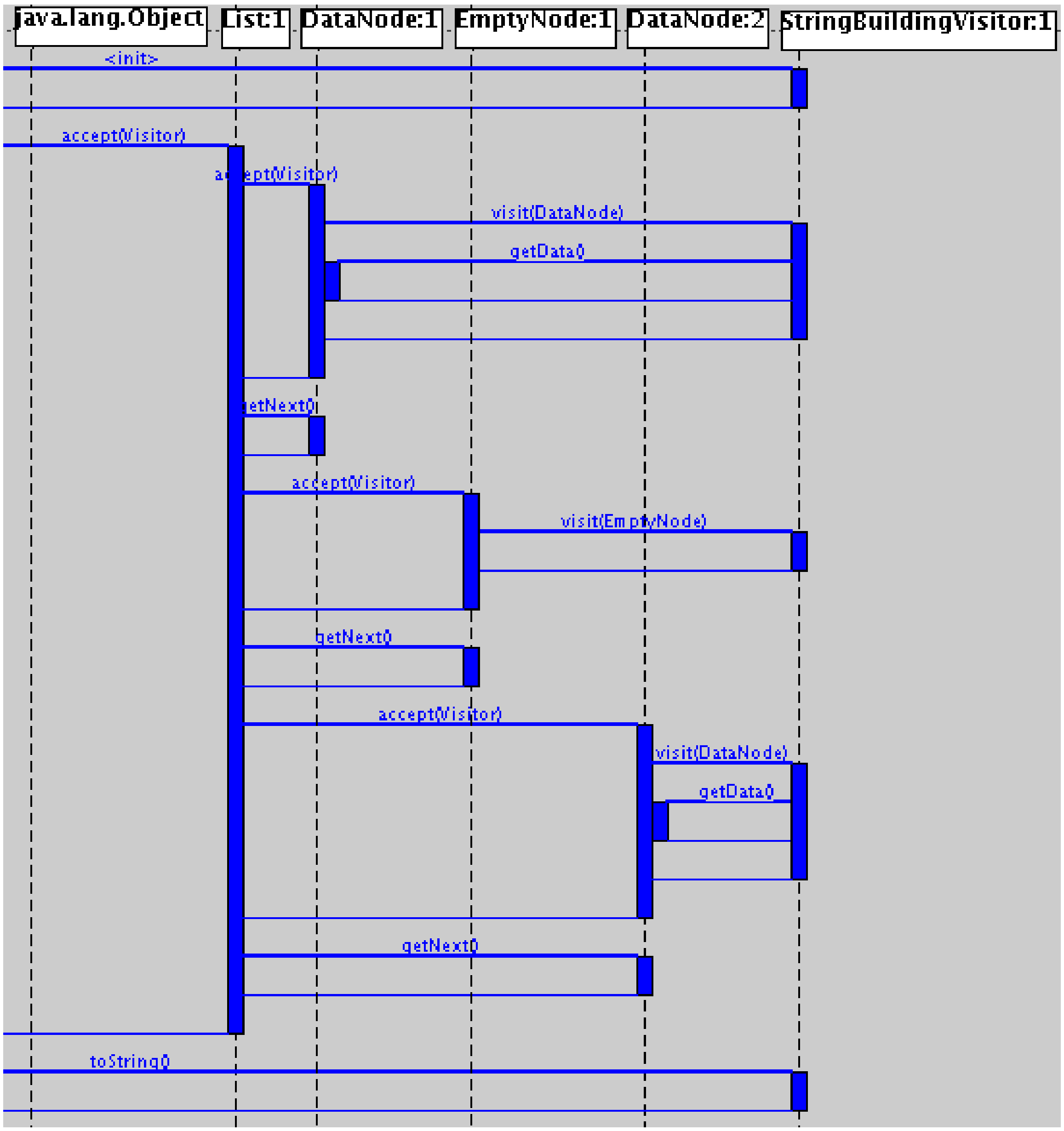


Figure 4.9: A JIVE-generated sequence diagram for an instance of the visitor pattern as implemented in Listing 4.4. The sequence diagram has been edited for space; not shown is are the static contexts, the `main` method, and the construction of the three-node list structure. The selected portion highlights the behavior of the visitor pattern: the `accept` method is called on `DataNode:1`, `EmptyNode:1`, and `DataNode:2`, in that order. The two data nodes use double-dispatch to collect their data, as illustrated by the calls from `StringBuildingVisitor:1` back to the data nodes; the implementation of visiting empty nodes does not require returning to the `EmptyNode` object.

```
class Driver {
  public static void main(String[] args) {
    List list = new List();
    list.addData("Hello");
    list.addEmpty();
    list.addData("World");
    Visitor v = new StringBuildingVisitor();
    list.accept(v);
    System.out.println(v);
  }
}
```

Listing 4.4: Visitor `Driver` implementation. This program will print the classic "Hello World" message.

the rectangles represent the focus of control, and an object lifeline only has a rectangle while a method executes in its context. This implies that if a horizontal "slice" of time is taken, any intersected rectangles will be on the lifelines of objects that have methods active at that point in time. Figure 4.9 illustrates an alternate paradigm, which we call the *method-call paradigm*. In this model, rectangles represent the entire duration of method calls, even if they are suspended while calling other methods. The implication here is that if a horizontal "slice" of time is taken, then the rectangles intersected at each lifeline represent the number of methods active (*i.e.* the size of the call stack) for that particular object. Sequence diagrams using both paradigms can be found in the literature, and so the choice is made by the diagram author. The focus-of-control paradigm highlights the points of activity within objects, whereas the method-call paradigm highlights the behavior of multiple method activations. In JIVE, we opt for the latter paradigm because it is more conducive to scrolling diagrams: when scrolling a focus-of-control sequence diagram, it can be easy to lose track of how many method calls there are on a particular lifeline, whereas this is explicit in a method-call sequence diagram.

4.4 Debugging with Sequence Diagrams

Sequence diagrams provide a convenient visualization of program history, and they can be used to find implementation errors by comparing the actual interactions with the desired ones. The previous example illustrated how sequence diagrams can be used as a pedagogic tool to understand the visitor design pattern, but beyond teaching applications, sequence diagrams can also be used for debugging. Since the history of interactions is explicit in a sequence diagram, a user can use the diagrams to compare actual program execution behavior with his or her mental model thereof.

Staying in the domain of design patterns, we will present an example taken from a student's work

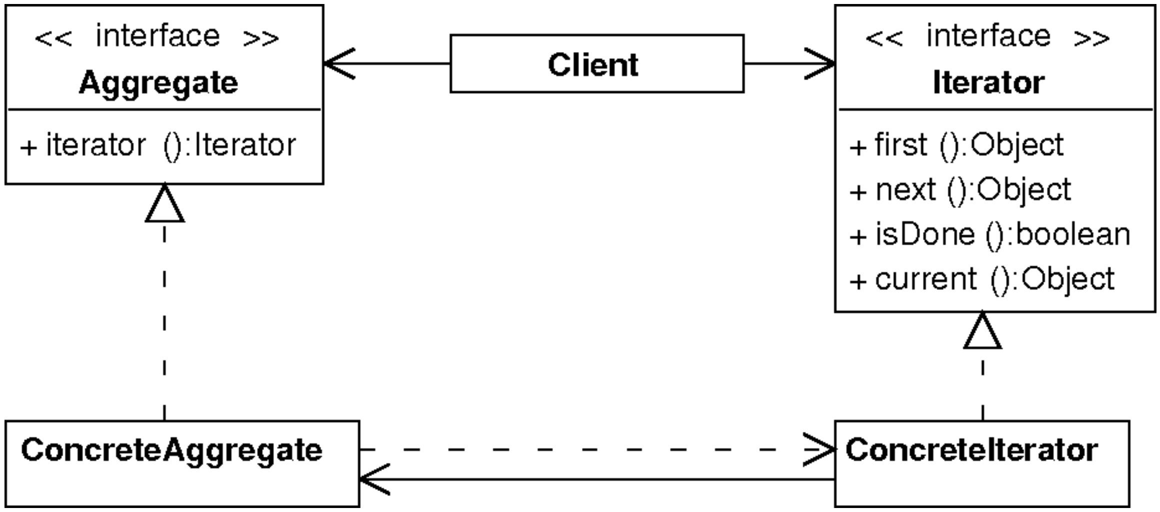


Figure 4.10: The architecture of the Iterator design pattern. This diagram is adapted from Gamma et al. [32].

on the *Iterator* design pattern. Iterator is a behavioral pattern that addresses the need for external iteration over an aggregate structure. Its solution strategy is that the aggregate itself returns an iterator object to the client, and the client then accesses the aggregated objects through the iterator's interface. The structure of the iterator pattern is shown in Figure 4.10 for reference.

A student was asked to demonstrate the visualization of the iterator design pattern using JIVE and so was required to implement an instance of the pattern. The student presented the sequence diagram of Figure 4.11 as a solution to the problem. To one who already understands the iterator pattern, this figure indicates that there is something seriously wrong with the student's implementation. It is expected that, during iteration, there would be interactions between the iterator and the structure over which it is iterating. However, the sequence diagram generated by the student shows no such interactions. It is interesting to note here that it is not a specific interaction that is missing, but rather a failure of the sequence diagram to conform to the visual *pattern* of iterator.

Listing 4.5 shows a portion of the student's faulty implementation of the iterator pattern. The bug here is not truly one of implementation but of design; the code demonstrates that the student did not understand what was required of the pattern. Upon creation, the "iterator" in this program passes through the entire tree using an inorder traversal and records the values in an array. Although this iterator will return the correct list of values, it does not comply with the specification of the pattern as given by Gamma *et al.*

A class diagram for the complete program of Listing 4.5 is shown in Figure 4.12. Comparing this architecture to Figure 4.10, which shows the correct iterator architecture, yields an interesting result:

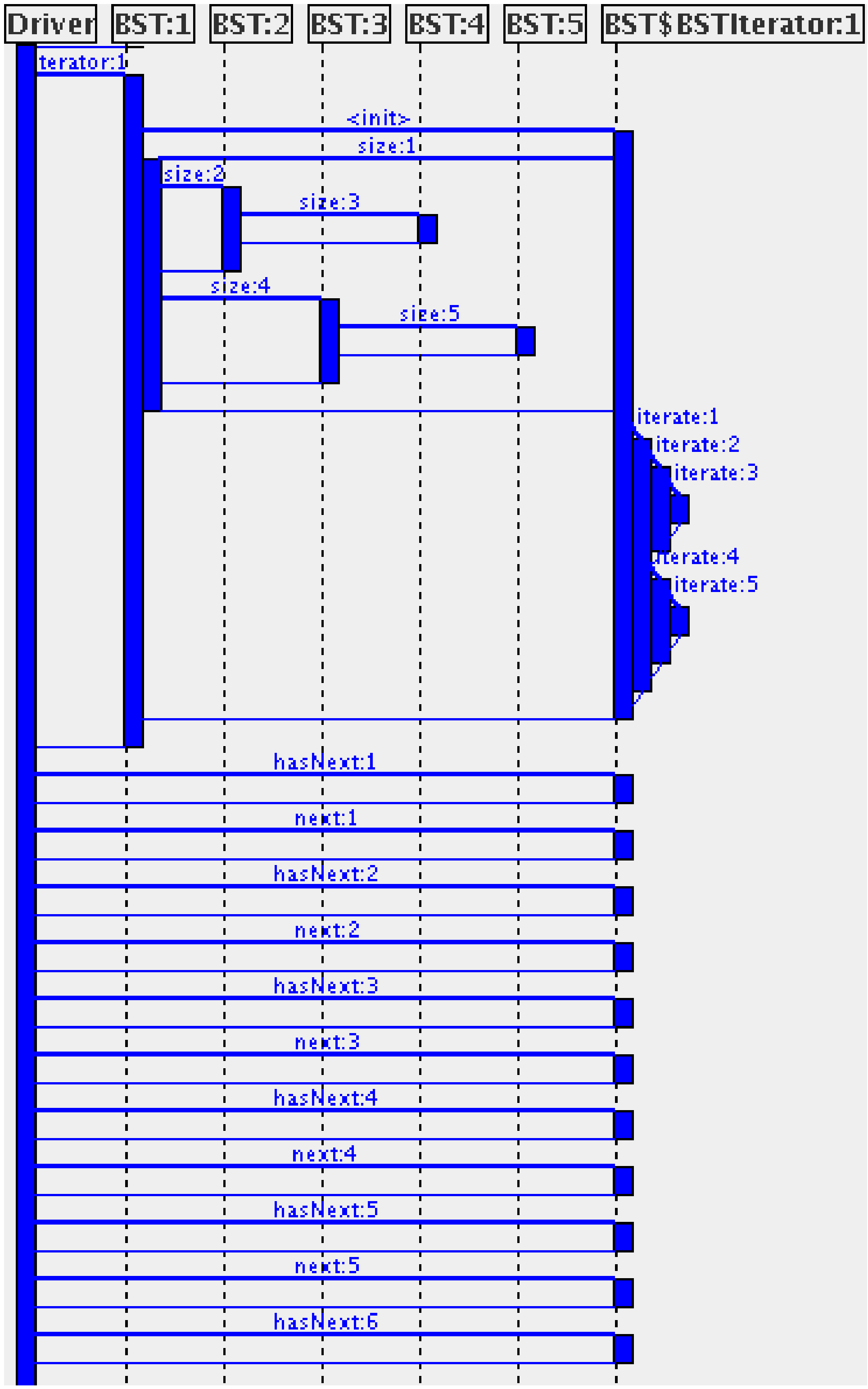


Figure 4.11: A JIVE-generated sequence diagram of a student's iterator implementation. The initialization of the tree has been hidden. The sequence diagram illustrates that there is a problem with the proposed solution, since a proper iteration interaction is not present.

```
class BST {
  private int data;
  private BST left, right;

  public BST(int data) {...}
  public void insert(int data) {...}
  public int size() { ... }

  private class BSTIterator implements Iterator {
    private int[] values;
    private int index;
    private BSTIterator() {
      BST root = BST.this;
      iterate(root, values, 0);
    }
    private int iterate(BST node, int[] store, int index){
      if (node.left!=null)
        index = iterate(node.left, store, index);
      store[index++] = node.data;
      if (node.right!=null)
        index = iterate(node.right, store, index);
      return index;
    }
    public boolean hasNext() { return index < values.length; }
    public int next() { return values[index++]; }
  }
}
```

Listing 4.5: Incorrect iterator implementation. The "iterator" in this program builds an array of values upon creation. This is not the correct behavior of an iterator according to the design pattern's definition, which specifies that the iterator should access the data structure on each `next()` request.

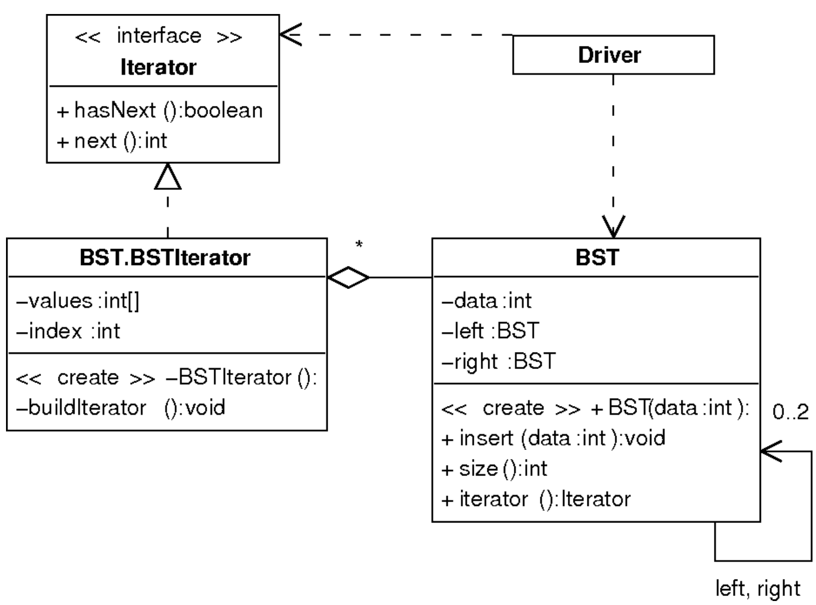


Figure 4.12: Architecture of the faulty iterator application

the student's solution architecture matches the textbook definition of the iterator architecture. In the student's architecture, `BST.BSTIterator` is the *ConcreteIterator*, `BST` is the `ConcreteAggregate`, and `Driver` is the *Client*. This phenomenon illustrates again the importance of dynamic runtime visualization, especially through sequence diagrams. Although the student's static architecture is correct, the dynamic behavior is wrong, as clearly indicated in the sequence diagram. Hence, the sequence diagram is not only useful for debugging, but it can clarify design errors that may be undetectable using static analysis. A sequence diagram generated by a correctly-implemented iterator can be found in Figure 4.13.

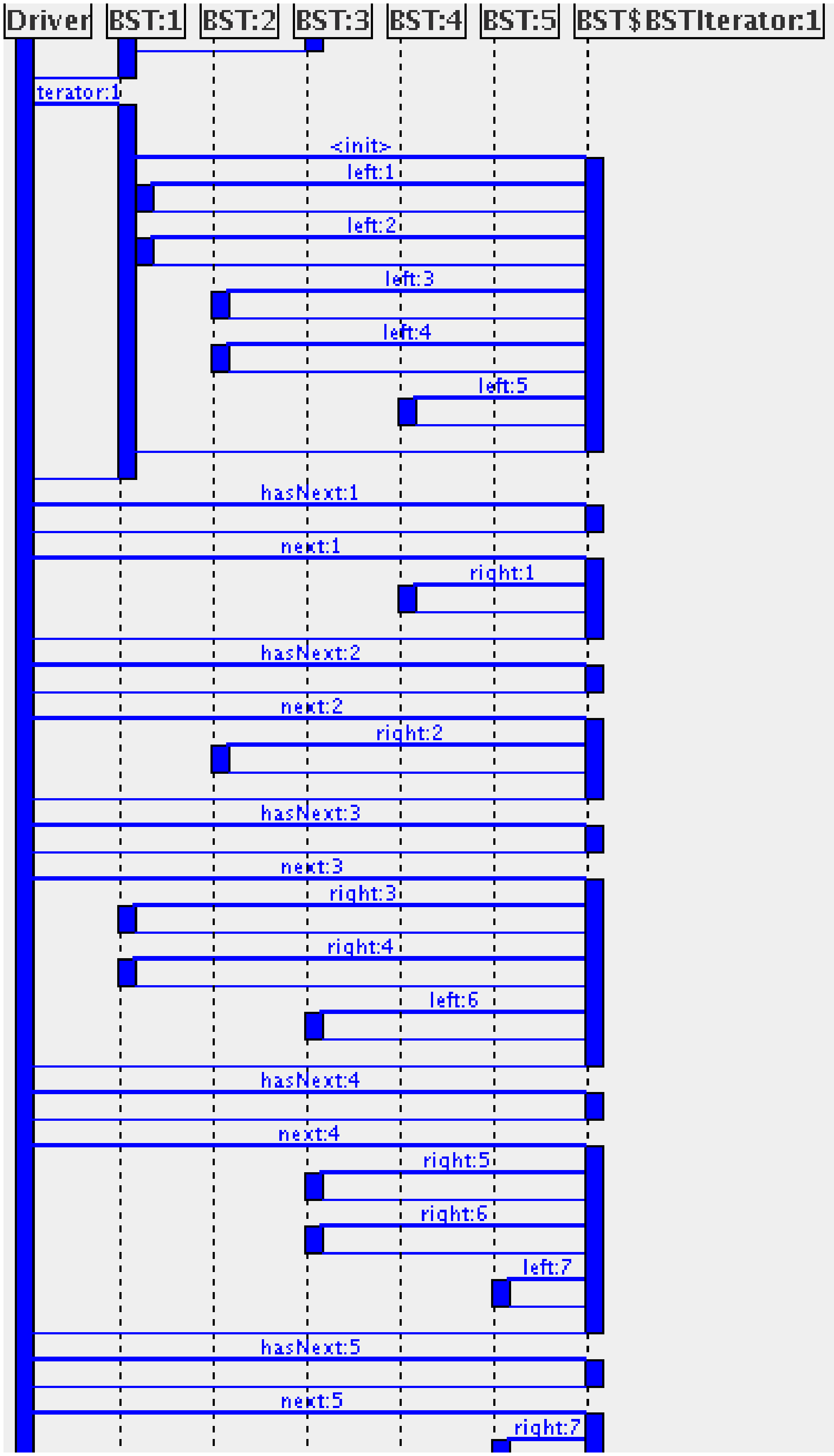


Figure 4.13: Sequence diagram for a proper iterator. Each time `next` is called, there are calls back to the tree nodes, indicating that, compared to Figure 4.11, this is a correct implementation of the iterator pattern.

Chapter 5

JIVE Architecture

This chapter provides an architectural overview of JIVE. Not all details of the implementation are supplied, but specific details are provided where novel techniques are used or where otherwise desirable. There is particular focus on JiveLog, the JIVE module that facilitates implementation of the contour model as well as interactive execution.

The remainder of this chapter is structured as follows:

- Section 5.1 provides a high-level description of the JIVE architecture. This includes a discussion of two approaches towards implementing interactive execution, one based on source-code transformation and the other based on using the Java Platform Debugger Architecture.
- Section 5.2 describes the JiveLog module, which includes the set of declarative events used to record program execution.
- Section 5.3 describes the JiveLogDB database schema, which provides the database layer for recording program execution.
- Section 5.4 presents our technique for instrumenting Java bytecode at runtime. This technique is required in order to collect the information necessary for building a JiveLog record.
- Section 5.5 describes how JIVE is integrated with the Java Platform Debugger Architecture.
- Section 5.6 presents the architecture for the sequence and contour models.

5.1 Overview and Motivation

Figure 5.1 provides a high-level view of JIVE's architecture. Each rectangle represents a module in the system, and the lines represent the possible paths of data flow between modules. "JPDA" is the

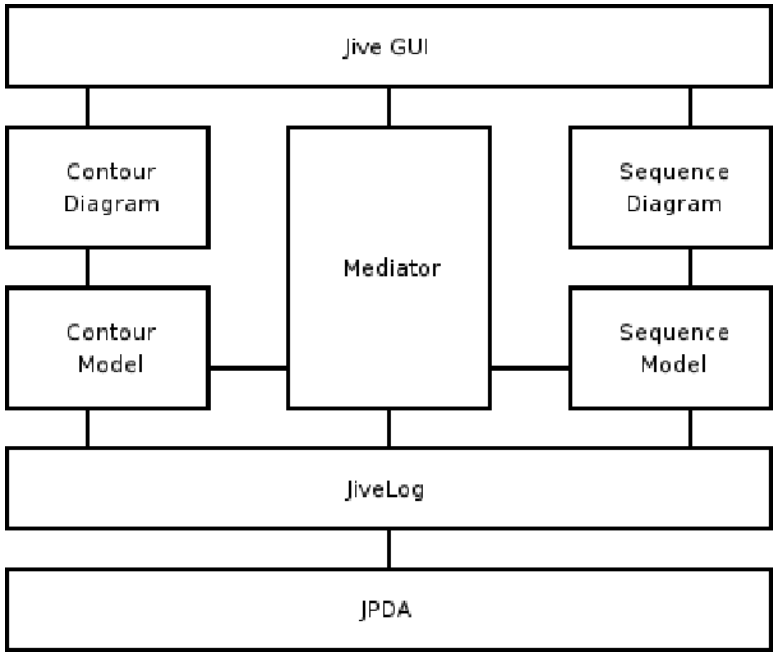


Figure 5.1: Architectural overview of JIVE

Java Platform Debugger Architecture, and it is the foundational library that enables JIVE. JPDA is used to initiate communication with and control the execution of the visualization client. It produces fine-grained, Java-specific execution events that are are then interpreted by JIVE into JiveLog events. The JiveLog module maintains a *contour-centric* representation of program history, described in section 5.2. The sequence and contour models, which store the history and state respectively, are perspectives of the JiveLog data, and the sequence and contour diagrams are visualizations of their respective models; in this way, the structure of the data model and visualization follows a model-view-controller architecture. The Mediator module facilitates communication between these different data models as well as the JIVE graphical user-interface.

There are two primary visualizations in JIVE: the visualization of program execution state using enhanced object diagrams and the visualization of program execution history using enhanced sequence diagrams. Program state visualizations depict object structure, instruction pointers, and the values of variables, whereas execution history visualizations display the activity of objects over time, specifically the messages passed among objects and the corresponding method activations. It is impossible to construct one of these visualizations solely from the other because the two visual paradigms contain different data. This relationship contrasts with other diagram types, for example, sequence and collaboration diagrams in the Unified Modeling Language: sequence and collaboration diagrams can be constructed from each other since each is an interaction diagram, presenting the same data in a different way. We can easily prove that this freedom of conversion is not present in state and history diagrams:

- A state diagram cannot be created from a history diagram since a sequence diagram does not contain information on variables' values. These values are integral to the construction of an enhanced object diagram.
- A history diagram cannot be created from a state diagram since a state diagram has no information about methods that have finished executing. This information is integral to the construction of a sequence diagram.

This relationship of non-conversion poses a dilemma to interactive visualization in a system such as JIVE. The two types of visualizations need to be integrated with one data model, but neither one by itself contains the proper information to build the other. There are two approaches to solving this dilemma: either embed extra information in one of the data models, allowing the other to be extracted from it, or use a separate, integrated data model from which both state and history diagrams can be drawn. The former approach is feasible, yet it can lead to unnecessary coupling and maintenance difficulties. The latter approach exhibits a preferable decomposition since the execution data is intrinsically in a separate form, and so there is no undisciplined embedding of impertinent data into semantically separate models. Additionally, having an essentially independent model facilitates future additions to the model without requiring modification to the state or history models, which are essentially views of the main model.

Source-Code Transformation Technique

A previous version of the JIVE contour visualization tool, detailed in a prior publication [37], did not include sequence diagram visualizations, and hence it was able to use the contour model as the primary data representation. That is, its design did not encounter the dilemma described above. Changes to the contour model were recorded as transactions, although in a different manner than the current JIVE architecture. This precursor to the modern JIVE used source-code transformation rather than the Java Platform Debugger Architecture. The source-code transformation technique involved loading a Java program's source, and then annotating the source to make it self-visualizing, as illustrated in Figure 5.2. Note that this approach is not used in the JIVE architecture, but it is useful to present this model in order that one understand the history and context of the JIVE methodology.

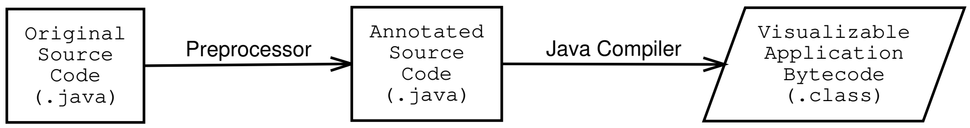


Figure 5.2: A data-flow diagram for the source-code transformation approach to interactive visualization. As the diagram indicates, the transformation itself is done by a preprocessor, and the resulting code is fed directly into a standard (non-modified) Java compiler. The result is a self-visualizing class file.

Figure 5.3 presents a control-flow diagram for the overall architecture of the source-code transformation approach. The client program is annotated so that important events are passed to a singleton `Mediator` object (not to be confused with JIVE's *Mediator* module), and these events are sent via Java Remote Method Invocation (RMI) to the visualization system. The visualizer then interprets these events and generates an object diagram depicting the program's state based on the received events. Since the RMI calls are synchronous, the client can be suspended until the visualization system acknowledges the call. The visualization system suspends return confirmation until the user requests that the client continue. This model is therefore a bidirectional communication although the data flow is essentially one-way, from client program to visualization system. Like the modern JIVE tool, this older design uses a transaction log to store states, and hence reverse execution in this approach is implemented as state recording and restoration.

A sample source transformation is shown in Figure 5.4. The transformation hinges on the definition of *inspectable* objects. The *Inspection API* is a custom library designed for the visualization tool, and it allows access to object, class, and method state by sending `Serializable` proxies across the RMI network connection. The source-code transformation algorithm itself can be summarized by the following steps:

1. Each class must acquire a reference to the `Mediator` singleton. The reference is kept in a variable that is not otherwise used in the program; by default, the variable "`_m`" is used.

2. Each class creates an inspectable proxy of itself. This proxy implements the `ObjectInspectable` interface, a part of the Inspection API. This proxy is stored in an unused instance variable, and the name "`_o`" is used by default.

3. For each method, create a `MethodInspectable` proxy, where `MethodInspectable` is another interface of the Inspection API. At the start and end of the method, send the method proxy to the visualization system through the mediator with a message describing the method's behavior.

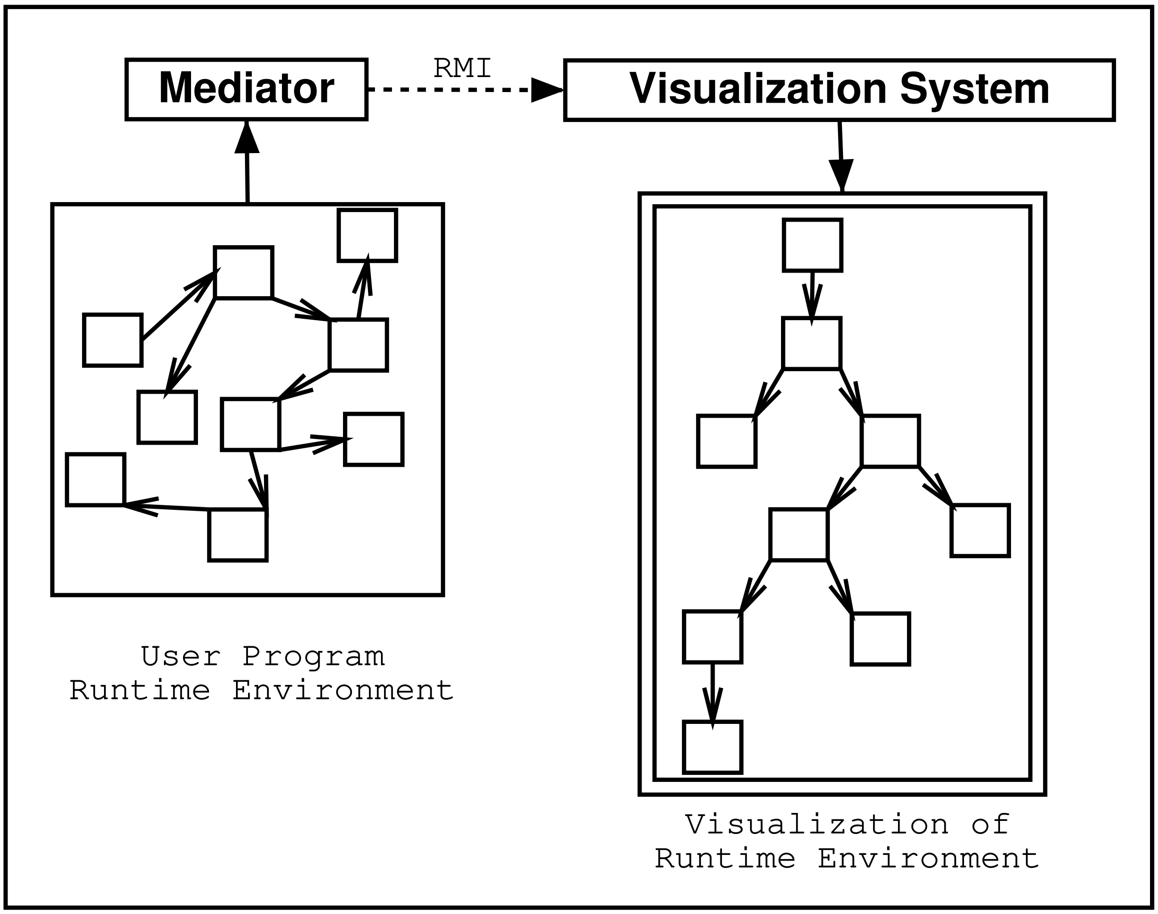


Figure 5.3: A control-flow diagram for the source-code transformation approach to interactive visualization. The `Mediator` sits between the user program and the visualization system, which is a separate process. Information regarding the activity of the user program is sent via Java RMI to the visualization system, which then generates contour visualizations of program state.

```
public class Tree {
 ...
 public void print(PrintStream s) {
  if (left!=null) {
    left.print(s);
  }
  s.print(value);
  if (right!=null) {
    right.print(s);
  }
 }
}
```

Source
transformation

```
public class Tree {
 static Mediator _m =
     Mediator.getSingleton();
 ObjectInspectable _o =
     new ObjectInspection(...);

 ...

 public void print(PrintStream s) {
   MethodInspectable _i =
     new MethodInspection(_o, ...);
   _m.methodStarted(_i);
   if (left != null) {
     _m.blockEntered(_i);
     left.print(s);
     _m.statementFinished(_i);
   }
   s.print(value);
   _m.statementFinished(_i);
   if (right != null) {
     _m.blockEntered(_i);
     right.print(s);
     _m.statementFinished(_i);
   }
   _m.methodFinished(_i);
  }
}
```

Figure 5.4: Sample source-code transformation from the source-code transformation approach to interactive program visualization. Each statement, method, and block are annotated with calls to the singleton mediator, which sends this information to the visualization system.

4. For each block, add block-entry and block-exit notification, using the inspectable method proxy defined above.

5. For each statement, add statement-started and statement-finished notification, using the inspectable method proxy defined above.

This approach adds significant overhead to the visualization client, requiring at least two remote method calls per statement, block, and method. However, the approach has a significant elegance in that the transformed program is still 100% Pure Java, and it can be compiled with any standard Java compiler. Additionally, any information that can be serialized can be passed between the client process and the visualization process through RMI. However, there are inherent complications with this approach, such as capturing exceptions that are thrown by library classes, which are not transformed. Another example is the handling of calls to superclass constructors; such calls must be the first statement within a constructor, and hence it is impossible to embed specialized code before the call using source-code transformation. These complications have led our design away from source-code transformation towards an approach grounded in runtime reflection through the Java Platform Debugger Architecture (JPDA). Where RMI is a general interprocess communication technology, the JPDA is tailor-made for debuggers and similar applications that require execution information from programs at runtime.

5.2 JiveLog Execution Events

In this section, we explore the format of JiveLog events and their application for representing program execution information. JiveLog is the module of JIVE that records and maintains a complete history of program execution. It is a singular model which contour models and sequence models (*i.e.* state and history models) can be formed. It also provides the interface through which the JIVE interactive query system can be implemented. We use the term "JiveLog" to refer to the module, its interfaces, and the format of serialized data; the referenced aspect is clarified when necessary. The JiveLog execution events are interpreted from JPDA execution events. The JPDA events themselves are read from the client program using the Java Debugger Interface (JDI), the Java API to the JPDA. However, there is not a one-to-one correspondence between JDI events and JiveLog events. The two layers have different design and motivation: JDI provides a JVM-centric view of program execution, and JiveLog provides a contour-centric view.

A Contour-Centric Representation

It is necessary that JiveLog contain sufficient program execution information to support online or offline visualization. *Online visualization* refers to visualization during client execution, and *offline visualization* refers to those constructed from recorded data. It is desirable for both types of visualization to be supported within the JIVE architecture since this provides the most freedom to the user. In order to support the offline visualization, JiveLog must maintain a *contour-centric* representation of program execution. This means that JiveLog incorporates such ideas as concrete and virtual contexts, static space, and the two types of inner objects. However, JiveLog itself is not a contour model; rather, it is a data format from which contour models can be constructed. JiveLog stores information about *contexts* rather than *contours*. The contour model, which is built from JiveLog data, interprets the contexts as contours. Separating contexts from contours improves the modularity of the system by reducing the coupling between the JiveLog and the contour model modules. For simplicity, we assume that JiveLog uses the same naming and number scheme for contexts as the contour model uses for contours, although in practice this is not necessary since the two could be resolved by a simple table look-up.

A method is defined once in a program, but it can be called many times. Each invocation can be represented by a method contour, and each of these contours will have a similar structure: they share member table definitions as well as source-code attachments. This prototypical form of a contour is represented as a *contour format*. There are object, method, and static contour formats for object, method, and static contours, respectively. All instances of a class share a contour format, and all activations of a method share a contour format; each static contour is the sole instance of its static contour format. Contour formats are a conceptual tool and do not have an explicit representation or encoding, and they are useful in the description and definition of JiveLog events. Specifically, using contour formats can save a great deal of space by reducing redundancy in the data model. For example, consider a case where a method m contains ten local variables; each activation of m contains these ten variables, and so in a naïve implementation, the execution log would contain all ten definitions for each activation. A much better solution is to use contour formats, so that each activation of the method need only reference its contour format, and so the ten variables of m are only defined once, within $m.cf$, the contour format for m. If v is the number of variables in a method m, and m is called n times during program execution, then the naïve solution takes $O(vn)$ space whereas the contour format solution takes $O(v) + O(n)$ space.

Type	Encoding	Example
`byte`	integer literal	"25"
`short`	integer literal	"25"
`int`	integer literal	"25"
`long`	integer literal	"25"
`float`	real literal	"3.14"
`double`	real literal	"3.14"
`char`	string literal	"x"
`boolean`	string literal	"false"
Reference type	contour name	"BST:3"

Table 5.1: String encodings of variable values

Value Encoding

Types in Java can be either *primitive* or *reference* types, a dichotomy that is an artifact of Java's C-style syntax. A primitive type holds a fixed range of values, and a reference type holds references to objects; Java's object references are an implicit form of pointers. The set of primitive types is composed of: the integer types `byte`, `short`, `int`, and `long`; the floating-point types `float` and `double`; the character type `char`; and the boolean type `boolean`. A variable whose type is not one of these is necessarily a reference type, and hence we can determine through static analysis whether a variable is of a primitive or reference type. Java's static typing therefore facilitatesusing static analysis to predict variable behavior at least with respect to the nature of the variable's type.

A reference variable that is not `null` must reference an object, which is represented in the contour model by an object contour. Recall that each object contour has a unique name with respect to the contour model. We can therefore encode variables' values using the following scheme: primitive values are encoded with their corresponding string literals, and reference values are encoded according to the name of the object contour they reference. For example, an `int` variable could have its value encoded as "417," and a `java.lang.Object` variable could have its value encoded as "Object:12," where this string is the name of the corresponding object's contour. The exact encodings for each type are listed in Table 5.1; note that not all of these encodings are equivalent to Java's literal encodings for the corresponding types (*e.g.* `float`, whose literals are annotated with an "f.").

JiveLog Events

Execution events are represented declaratively in JiveLog. Each event has a unique sequence number that reflects the order in which events occurred in the client program. The events encapsulate unit changes in program state. We define each event as a tuple below.

- $LOAD = (time, cid, class, parent, thread)$ represents the loading of a class, where: *time* is the sequence number representing the time at which the class is loaded; *cid* is the context identifier for the new class context; *class* is the name of the class that was loaded; *parent* is a context identifier of the parent (enclosing) static context, if any; and *thread* identifies the thread on which the event took place.

- $NEW = (time, I, C, thread)$ represents the creation of an object, where: *time* is the sequential time when the object was created; I is a sequence of n context identifiers $(cid_0, cid_1, \ldots, cid_n)$, where cid_0 through cid_{n-1} are for virtual contexts and cid_n is for the concrete context; C is a sequence of class identifiers $(class_0, class_1, \ldots, class_n)$, where $class_i$ is the name of the class for for each $cid_i \in I$; and *thread* identifies the thread on which the event took place.

- $CALL = (time, name, A, cid, context, thread, caller, source, line)$ represents the calling of a method, where: *time* is the sequential time of the method call; *name* is the name of the method being called; A is the sequence of encoded actual parameters $actual_0$, $actual_1$, …, $actual_n$; *cid* is the context identifier for the newly-called method; *context* is the context identifier of the context in which the method should be placed; *thread* identifies the thread on which the call is made; *caller* is the context identifier of the calling method or null if the caller is unknown; *source* identifies the source code file containing the method; and *line* is a line number of *source* where the method definition starts.

- $RETURN = (time, cid, thread)$ represents a normal method return, where: *time* is the sequential time of the return; *cid* is the context identifier of the method that has returned; and *thread* identifies the thread on which the event took place. The thread encoding is somewhat redundant here, since the *cid* can be used to identify the matching CALL event, and a thread must return on the same thread on which it is called. However, representing the thread here provides a convenient parallel among event definitions, and it allows the thread to be determined practically instantly rather than requiring indirection (*i.e.* a look-up of the matching CALL event.).

- $STEP = (time, thread, source, line)$ represents a change in source code line, where: *time* is the sequential time of the step; *thread* identifies the thread that has stepped; *source* identifies the source code of the class in whose context the step has taken place; and *line* refers to a line of source code within *source*.

- $SET = (time, name, context, value, thread)$ represents a change in a variable's value, where:

time is the sequential time at which the change took place; *name* is the name of the variable that changed; *context* is a context identifier referencing the context that contains the variable whose value changed; *value* is the (encoded) new value of the variable; and *thread* identifies the thread on which the event took place.

- $EXCEPTION = (time, object, thrown, caught, source, line, thread)$ represents an exception's being thrown, where: *time* is the sequential time at which the exception was thrown; *object* is the context identifier for the exception object itself; *thrown* is the context identifier for the method that threw the exception; *caught* is the context identifier for the method that caught the exception; *source* identifies the source code of the file where the exception was caught; *line* is the line of *source* where the exception was caught; and *thread* identifies the thread on which the event took place. As in the RETURN event, *thread* here is somewhat redundant since it can be determined by cross-referencing *thrown* or *caught*, but encoding it here provides a convenience to and parallelism in the model.

The CALL, STEP, and EXCEPTION events each include information about the source file and line number where the event took place. This information is needed in order to support source-code highlighting and source-based queries, but this can lead to large amounts of redundant storage. Similarly, the STEP event itself is not strictly necessary for simple visualizations; it is only necessary to provide source-related features such as highlighting, breakpoints, and queries. This data can be omitted to save space, although the current implementations assume that full logs are used.

5.3 JiveLogDB Database

The `JiveLog` interface of the JIVE software architecture is designed so that the implementation can vary. This loose coupling of components promotes a modular and extensible software architecture. In the current prototype, JiveLog events are recorded in memory using a Java `ArrayList` Collection. The advantage of this approach is simplicity, but clearly it cannot scale to long-running programs. The in-memory JiveLog module can be replaced with *JiveLogDB*, a module that stores events in files or in databases. Using a database to represent program execution is a significant and novel idea, and it is important to the JIVE methodology for several reasons. Storing events in a database facilitates offline analysis of program logs or review of previously-visualized program executions. Also, under our model of *debugging as querying*, JIVE queries can be encoded as queries over the relational database, which leverages the strength of the database for the domain of program visualization.

Hence, we can leverage the power of relational database towards the new application domain of interactive visual debugging with JIVE.

There are two types of tables in the JiveLogDB database schema: contour format tables and event tables. *Contour format tables* contain the declarations of local variables (attributes), methods, and inner classes defined within program units; that is, they contain the data that comprises a contour format. *Event tables* contain actual program execution events, and these database tuples encode JiveLog events. Every event recorded in an event table uses a *sequence number* as a primary key. The sequence number is an integer corresponding to the order in which events are executed. No two JiveLog events have the same sequence number, and so using the sequence number as a primary key allows us to define each type of JiveLog event in its own table.

Figure 5.5 presents the database schema for the JiveLogDB tables. The tables themselves are described below, with the descriptions separated into contour format tables and event tables.

Contour Format Tables

The contour format tables are specified below. Foreign keys are described by their referenced table and column. These contour format tables can be constructed by a static analysis of program source code, or they can be built using JPDA's access to Java's reflection capabilities.

CF: Contour format list.

CF (Primary key): A unique identifier for a contour format. The actual encoding of this identifier is essentially arbitrary; descriptive names can be used, although numerical identifiers can be used for more efficient database access.

NAME: A descriptive name of this contour format. This value is primarily to aid in debugging.

VARIABLES: Variable declarations.

ID (Primary key): A unique identifier for this variable declaration.

NAME: Name of the variable.

TYPE: Type of the variable.

METHODS: Method declarations.

ID (Primary key): A unique identifier for this method declaration.

NAME: Simple name of this method, without return type, formal parameters or parentheses.

SOURCE: Identifies the source file that declares this method.

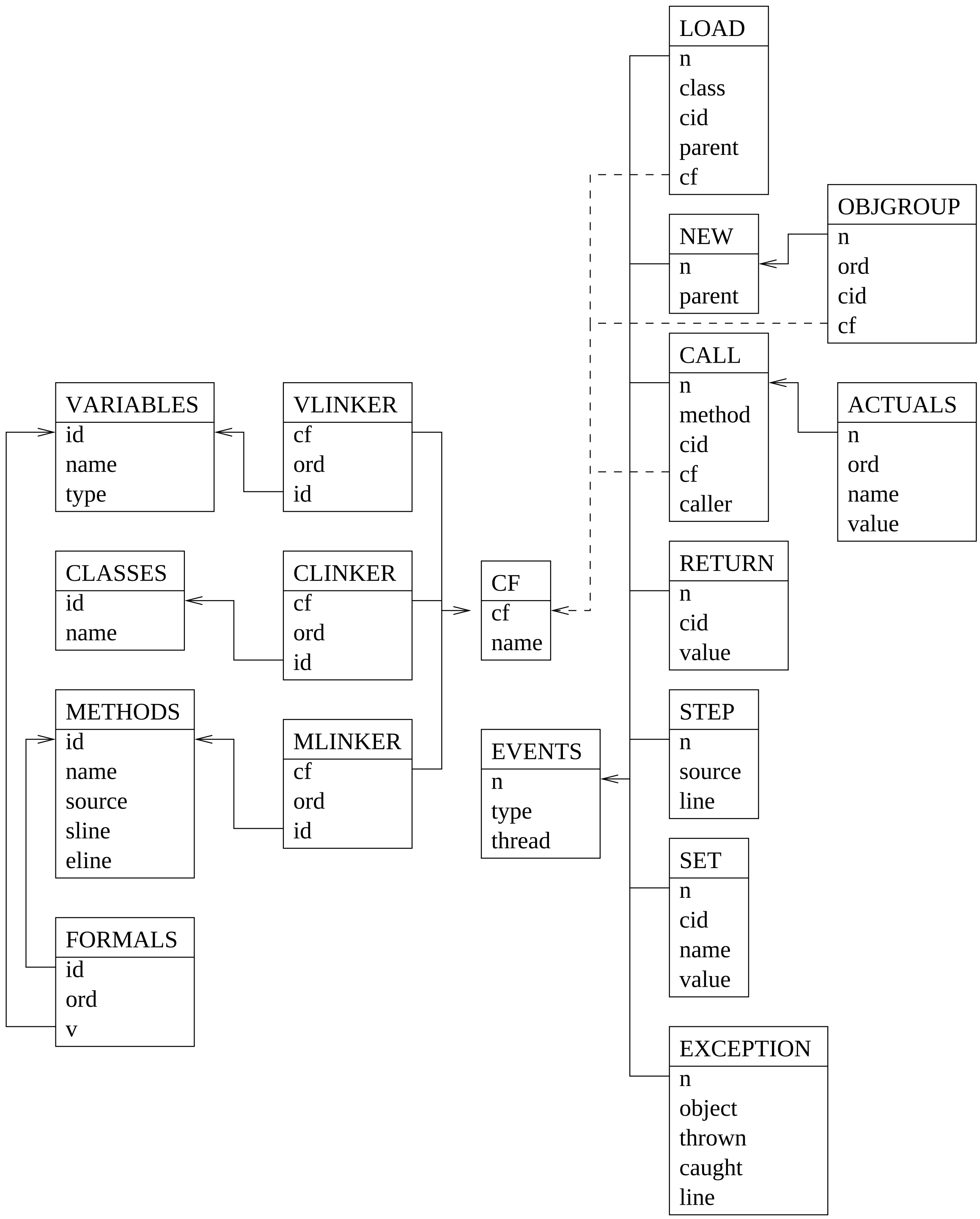


Figure 5.5: JiveLogDB table schema. The arrows indicate data dependencies via foreign keys. The tables on the left are the contour format tables, and those on the right are the JiveLog event tables. The dashed dependency arrows illustrate the dependencies between JiveLog events and the contour format data.

SLINE: Identifies the first line of this method declaration in the source file.

ELINE: Identifies the last line of this method declaration in the source file.

FORMALS: Formal parameters of a method declaration.

ID (Composite key): Foreign key of METHODS.ID, identifies the method declaration for which this is a formal parameter.

ORD (Composite key): Ordinal position of this formal parameter declaration in its list.

V: Foreign key of VARIABLES.ID, describes the variable that is the formal parameter.

CLASSES: Inner class declarations.

ID (Primary key): A unique identifier for this inner class declaration.

NAME: The name of the class.

VLINKER: Links contour formats to variable declarations.

CF (Composite key): Foreign key of CF.CF, identifies the contour format to which the variable is linked.

ORD (Composite key): Ordinal position of this variable declaration within the contour format.

ID: Foreign key of VARIABLES.ID, identifies the variable.

CLINKER: Links contour formats to inner class declarations.

CF (Composite key): Foreign key of CF.CF, identifies the contour format to which the inner class is linked.

ORD (Composite key): Ordinal position of this class declaration within the contour format.

ID: Foreign key of CLASSES.ID, identifies the inner class.

MLINKER: Links contour formats to method declarations.

CF (Composite key): Foreign key of CF.CF, identifies the contour format to which the method is linked.

ORD (Composite key): Ordinal position of this method declaration within the contour format.

ID: Foreign key of METHODS.ID, identifies the method.

Log and Event Tables

The following descriptions are for JiveLogDB log and event tables. The EVENTS table represents the log of all events, and the other tables encode those events using the same nomenclature as their JiveLog counterparts.

EVENTS: List of JiveLog events.

N (Primary key): Sequence number of this event.

TYPE: The type of the event; one of "LOAD," "NEW," "CALL," "RETURN," "STEP," "SET," or "EXCEPTION."

THREAD: The long integer identifier of the thread on which this event occurred. Note that although JiveLog's mathematical representation encodes the thread with each event object, it is refactored into this separate table here for improved normalization.

LOAD: JiveLog LOAD events.

N (Primary key): Sequence number of this event; foreign key of EVENTS.N.

CLASS: The name of the class that was loaded.

CID: The context identifier of the static context corresponding to this class, which is the context created by this event.

PARENT: The context identifier (cid) of the context that encloses this event's static context; this can be the reserved value "$NONE" if and only if the static context is not nested in another context.

CF: Contour format of the static context; foreign key of CF.CF.

NEW: Jivelog NEW events.

N (Primary key): Composite key; sequence number of this event; foreign key of EVENTS.N.

PARENT: Context identifier of the enclosing context for the newly created object group. This is only used if an inner object is being created; otherwise, the reserved value "$NONE" is used.

OBJGROUP: Object groups created by NEW events.

N (Composite key): Sequence number of the corresponding NEW event. Foreign key of NEW.N.

ORD (Composite key): Ordinal position of this object within its object group.

CID: Context identifier of the newly created context.

CF: Contour format of the new context; foreign key of CF.CF.

CALL: Jivelog CALL events.

N (Primary key): Sequence number of this event; foreign key of EVENTS.N.

METHOD: Foreign key of METHOD table identifying this method.

CID: The context identifier of the method context created by this call.

CF: The contour format of this method; foreign key of CF.CF.

CALLER: Context identifier of the caller of this method; May be reserved value "$SYSTEM" if and only if this is the first method on its thread.

RETURN: Jivelog RETURN events.

N (Primary key): Sequence number of this event; foreign key of EVENTS.N.

CID: The context identifier of the method that returned.

VALUE: The encoded return value of the method.

STEP: JiveLog STEP events.

N (Primary key): Sequence number of this event; foreign key of EVENTS.N.

SOURCE: Identifies the source file in which the step occurred.

LINE: The line number location of the step.

SET: JiveLog SET event.

N (Primary key): Sequence number of this event; foreign key of EVENTS.N.

CID: Context identifier of the context whose variable changed.

NAME: The name of the variable that changed.

VALUE: Encoded new value of the variable.

EXCEPTION: JiveLog EXCEPTION event.

N (Primary key): Sequence number of this event; foreign key of EVENTS.N.

OBJECT: Context identifier of the exception object itself; may be the reserved value "$UNKNOWN" if it is unavailable.

THROWN: Context identifier of the method that threw the exception; may be the reserved value "$UNKNOWN" if the thrower is not in the model.

CAUGHT: Context identifier of the method that caught the exception; may be the reserved value "$NONE" if the method is uncaught.

LINE: The line number at which the exception was caught; this is only meaningful if CAUGHT is not "$NONE."

ACTUALS: List of actual parameters of a method.

N (Composite key): Foreign key of CALL.N, specifies the method call for which these are actual parameters.

ORD (Composite key): Ordinal position of this parameter in the list.

NAME: The name of the formal parameter.

VALUE: The encoded value of the actual parameter.

5.4 Bytecode Instrumentation with a Custom Classloader

The Java Platform Debugger Architecture is a set of tools and specifications that are intended for use by Java debuggers. It provides access to nearly all of the information required to generate JiveLog execution data and therefore JIVE visualizations. However, the JPDA does not provide any mechanism for determining the return value of a method. This information is not strictly required for state or history visualizations, but it is extremely desirable for interactive queries. Without this information, no queries can be posed over JiveLog that have to do with method return information, but such queries are an integral part of the plan for JIVE's query capability. Hence, it is desirable to incorporate some means of obtaining method return values in the JIVE architecture.

The Apache Software Foundation's Byte Code Engineering Library (BCEL)[1] is a Java library that facilitates the analysis, creation, and modification of Java bytecode. Java bytecode forms the instructions that are interpreted by the Java Virtual Machine, and they are usually found in compiled class files, that is, those files that end with ".class". One application domain of BCEL is *bytecode instrumentation*, which is the modification and augmentation of bytecode without altering a program's source code. This instrumentation can be done before a program is executed, but an even more powerful application is the modification of bytecode when a class is loaded. Normally, a Java program is compiled into bytecode (a class file), and the bytecode is loaded by a Java class loader when the corresponding class is needed. However, the default class loader can be replaced with any class that extends Java's abstract `java.lang.ClassLoader` class. The technique of bytecode instrumentation then involves replacing the default class loader so that when a class is loaded, its

[1] `http://jakarta.apache.org/bcel`

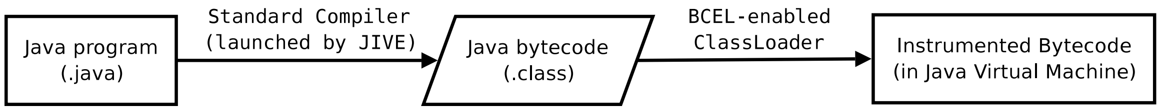


Figure 5.6: A data-flow diagram illustrating how JIVE uses a custom class loader. The original program is compiled with a standard Java compiler, and the result is a class file. The class file is loaded by a custom classloader, which alters the bytecode dynamically to add extra functionality to the class.

bytecode is dynamically modified immediately before passing the code to the JVM. This approach is used in JIVE, and an overview of the process is shown in Figure 5.6.

The approach used to extract method return values is inspired by the technique used in *Jassda*[2], an open-source tool for checking trace- and time-assertions in Java programs. The technique hinges on the fact that the JPDA does not support direct access to method return values, but it can be used to detect assignments to arbitrary variables. Additionally, although line number information is reported by JPDA, the debugger architecture does not work with program source code at all, but only with compiled class files that are annotated with debug information (*e.g.* compiled with the "`-g`" flag). The approach then involves modifying the bytecode of non-void methods so that immediately prior to returning a value, the value is stored into a local variable. This variable is called "`$return$`," a specially chosen name since it is technically not a valid Java identifier name and therefore will not clash with existing variable names (assuming no piggybacking of bytecode-instrumenting classloaders). Since the modification is made in the bytecode and not in the source code, the technically invalid name is never passed through the compiler, and so no errors are generated. The bytecode is further altered so that the method returns the value of `$return$` rather than its original expression, thereby avoiding recomputation and other undesirable side-effects. JIVE then uses JPDA to monitor for changes to local variables as usual, but when there is an assignment to a variable called `$return$`, we know that a method is about to return. Furthermore, we know that the return value for the method is stored in `$return$`. Listing 5.1 provides a Java class that computes Fibonacci numbers. Listing 5.2 shows how the source would look after bytecode modification. The source is not in fact modified, but the listing shows where the modifications are made at the bytecode level.

[2] `http://jassda.sourceforge.net`

```
public class Fibonacci {
  /**
   * Computes the n-th Fibonacci number recursively.
   * Returns -1 on invalid input.
   */
  public static int compute(int n) {
    if (n <= 0) return -1;
    else if (n <= 2) return 1;
    else return compute(n) + compute(n-1);
  }
}
```

Listing 5.1: Simple Java program to compute Fibonacci numbers.

```
public class Fibonacci {
  /**
   * Computes the n-th Fibonacci number recursively.
   * Returns -1 on invalid input.
   */
  public static int compute(int n) {
    int $return$;
    if (n <= 0) {
      $return$ = -1;
      return $return$;
    }
    else if (n <= 2) {
       $return$ = 1;
       return $return$;
    }
    else {
      $return$ = compute(n) + compute(n-1);
      return $return$;
    }
  }
}
```

Listing 5.2: Modified version of the `Fibonacci` class of Listing 5.1 to illustrate how method return values are computed. No source modification is actually performed; this modification is made directly to the bytecode by a custom classloader. This listing shows how the equivalent source would look.

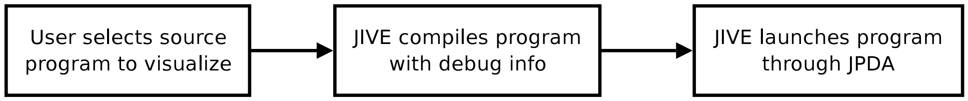


Figure 5.7: Data flow diagram for the loading, compilation, and execution of the visualization client within JIVE.

5.5 Integration with Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA) is an architecture designed primarily to support the development of debuggers, although the technology can be used in other domains, such as JIVE's interactive program visualization. The JPDA consists of two application programming interfaces (APIs) and two software components that integrate them, a front-end and a back-end. The two APIs are the Java Debug Interface (JDI) and the Java Virtual Machine Tool Interface (JVMTI).[3] JVMTI is implemented within the Java Virtual Machine, and the client is the JVM itself. JDI is a pure Java interface to JPDA, which is the back-end, and the client is the debugger. The Java Debug Wire Protocol (JDWP) defines the format of data passed between the debuggee and the debugger. We use Sun Microsystem's reference implementation of the JPDA, which requires that Sun's `tools.jar` file be used on the classpath of JIVE; this file is distributed with the Java Standard Edition Development Kit, but not with the Java Standard Edition Runtime Environment.

The JiveLog design leverages the flexibility of the JPDA in a two-process architecture. The visualization environment itself runs in one process, while the program being visualized, the *visualization client*, runs in a separate process. Interprocess communication is managed by the JDWP through JDI. The user provides a program to JIVE, and the program is compiled with the proper flags to generate debugging information. This program is then executed through JPDA; launching the client through JPDA ensures that the proper, platform-specific JDWP parameters are used. This compilation and execution procedure is outlined in Figure 5.7. In order to guarantee source-code highlighting functionality, a program must be loaded from its source code, but JIVE is capable of running visualizations without source-code highlighting by loading precompiled class files; this functionality requires that the class files contain debugging information, as generated by passing the "`-g`" flag to the standard Java compilers.

Once JIVE has started the client process, it registers listeners via JPDA and awaits notification. This subscription is implemented using JDI `EventRequest` objects. When the client generates a tracked event, the client's 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

[3]JVMTI has replaced the older JVMDI (Java Virtual Machine Debug Interface) as of Java 2 Standard Edition 5.0.

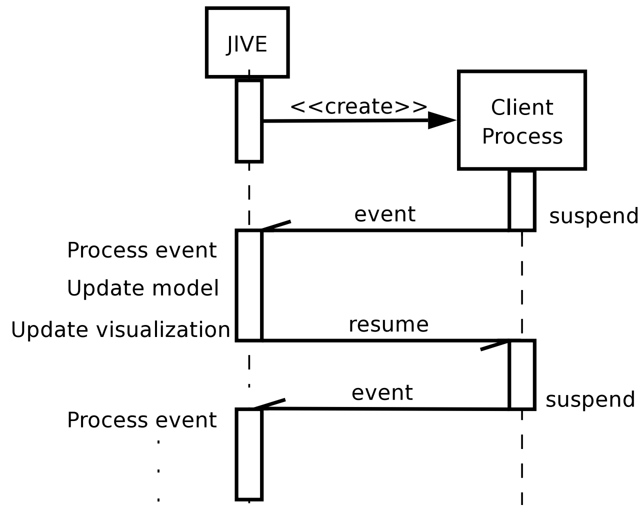


Figure 5.8: Overview of the interaction between JIVE and the visualization client. JIVE and the client run on two separate processes, shown as lifelines in the sequence diagram. We are able to provide interactive forward execution using the JPDA suspension and breakpoint functionality.

client program and then listens for subsequent events. An overview of this interaction is provided in the sequence diagram of Figure 5.8. How JIVE reacts to JPDA events is partially determined by its current *step policy*. The step policy specifies the size of steps and whether JIVE should synchronize client execution with JIVE's interactive stepping capability. Steps, as defined by the JPDA, can be individual operations or lines of source code. If the client is set to suspend, then its execution is paused while JIVE manages its own model and the visualizations thereof; the client is not suspended until the user directs it. The alternative is that the client is not suspended, in which case it executes continually as JIVE collects execution information. Complete details on this interaction mechanism are presented in Chapter 6.

Interpreting JDI Events

Some JiveLog events are direct interpretations of JDI execution events, but others require substantial analysis in order to compute. Figure 5.9 illustrates the relationship between JDI events and their corresponding JiveLog events. A brief description of each JDI event used by JIVE is provided below; these are all taken from the `com.sun.jdi.event` package.

- A `MethodEntryEvent` corresponds to a method's entry. This event is directly used to generate CALL events, and it is also used to infer when classes are loaded (LOAD events) and when objects are created (NEW); see below for details.

- A `MethodExitEvent` corresponds to a method's normal exit, that is, a non-exceptional return.

This event is used to generate RETURN events.

- A `StepEvent` corresponds to a step in the JVM. The step size is set by JIVE through JDI to be single lines of program execution. StepEvents are used to generate STEP events, but they are also necessary for monitoring a method's local variables, as discussed below.

- A `ModificationWatchpointEvent` corresponds to a change to a watched field. JIVE sets watchpoints on all fields, and so these events are used to generate SET events for fields (*i.e.* variables defined in a class but not in a method).

- An `ExceptionEvent` corresponds to an exception's being thrown and possibly caught, and hence these events can be used to generate EXCEPTION events.

Special Situations

There are three aspects of program execution that must be captured by JiveLog but that are not represented in JDI events. These are object creation, the updating of local variables' values, and capturing the return values of methods. Each of these is described below along with the solution strategy used in JIVE.

- *Object creation.* There is no JDI event that corresponds exactly to the creation of objects. JiveLog infers object creation and therefore NEW events from the invocation of constructors. Within the JVM, a constructor call is represented by the invocation of the initializer, `<init>`, which handles object initializers, field initializers, and the constructor code. When such a method invocation is intercepted by JiveLog, an appropriate NEW event is created and logged prior to the notification of the CALL of `<init>`.

- *Local variable updates.* Watchpoints can be used to monitor the changes to fields; this is done using JDI `ModificationWatchpointRequest` objects. However, there is no JPDA watchpoint mechanism for local variables. The changes to local variables can only be determined through successive comparison of stack frames when processing JDI `StepEvent` notifications. As of this writing, the approach taken in JIVE involves recording the last known stack frame for each thread. When a step occurs, the new frame is compared with the old, and if there are any discrepancies, SET events are fired for the local variables, and the last known stack frame is overridden with the new one. This approach takes $O(|V|)$ time, where V is the set of local variables of a method, including formal parameters. It is possible that static analysis using

JiveLog Event	**Sequence Model Effect**
LOAD	Create a lifeline
NEW	Create a lifeline
CALL	Start a method activation
RETURN	End a method activation
EXCEPTION	Possibly terminate one or more method activations

Table 5.2: JiveLog events and their effects on a sequence model

object-oriented slicing, such as the techniques used by Chen and Xu [20], may be used to reduce the set of variables that must be considered at each step.

- *Method return.* JPDA does not support accessing the return values of methods, and so we use the technique discussed above in Section 5.4, which involves using dynamic bytecode instrumentation through a BCEL-backed custom classloader.

5.6 Sequence and Contour Models

The sequence model maintains the data required to draw enhanced sequence diagrams. It specifies *lifelines* and the *method activations* on those lifelines. The *object* atop each lifeline corresponds to Java object or static contexts (for the activation of instance and static methods, respectively). Like JiveLog, a sequence model is essentially stateless: it is a declarative recording of events. In this sense, the sequence model is a simplified version of the JiveLog data. Representing the sequence model explicitly decreases the coupling between modules compared to an approach where the sequence diagram is drawn directly from JiveLog data. This explicit representation also facilitates faster processing of sequence data, which includes processing for drawing and for data mining (*e.g.* queries over program execution that can be answered with only sequence data). The sequence model should not be confused with a sequence diagram; under a model-view-controller architecture, a sequence diagram is a view of a sequence model.

The sequence model is built from a subset of JiveLog events as described below and summarized in Table 5.2:

- *LOAD*: When a class is loaded, a lifeline is created in the sequence model that corresponds to the class' static context. This lifeline will contain the static method activations for that class.

- *NEW*: When an object is created, a lifeline is created for it in the sequence model. Although JiveLog is aware of virtual and concrete object contexts, these different contexts coalesce into a single lifeline. This is because the sequence diagram is designed to clarify the sequence of method activations, not the precise semantics of the method itself — this is handled by the visual operational semantics for Java described in Chapter 3.

- *CALL*: A method activation is created on the appropriate lifeline of the sequence model when a CALL event occurs.
- *RETURN*: Method activation rectangles are terminated at the point corresponding to a JiveLog RETURN event.
- *EXCEPTION*: When an exception causes a change in object (or static) context, this must be reflected in the sequence model. However, if an exception is thrown and caught within the same method, there is no modification of the sequence model and hence no corresponding annotation on the sequence diagram.

SET and *STEP* events are not used in the creation of the sequence model.

Sequence Graph Representation

A *sequence graph* is a mathematical representation of a sequence model. It is convenient to have such a representation in order to formalize the syntax and semantics of the model. A sequence graph is defined as $S = (L \cup M, \mathbf{E}, start : M \rightarrow \mathbf{N}, end : M \rightarrow \mathbf{N} \cup \lambda, context : M \rightarrow L)$, where:

- L is the set of lifelines;
- M is the set of method activations;
- $\mathbf{E}$ is a set of directed edge sets E_i, $1 \leq i \leq |\mathbf{E}|$, where $E_i = C_i \cup R_i$;
- C_i is the set of edges (m_j, m_k), $m_j \in M$, $m_k \in M$, such that m_j calls m_k on thread i;
- R_i is the set of edges (m_k, m_j), $m_j \in M$, $m_k \in M$, such that m_k returns to m_j on thread i;
- *start* maps methods to their starting times;
- *end* maps methods to their ending times or to λ if there is no known ending time for the method; and
- *context* maps methods to their lifeline context.

It is necessary to have a sentinel λ to represent those methods whose ending times are not known. Sequence models are generated dynamically during program execution, and so it will frequently occur that a method has started but has not yet ended. We define a function $End : \mathbf{S} \rightarrow \mathbf{N}$ (where $\mathbf{S}$ is the set of sequence graphs) that given a sequence graph S returns the maximum value of *end* over all methods of S; when rendering a sequence diagram, it is convenient to assume that $\lambda = End(S)$, in which case the method is drawn to the current "end of time."

A sequence graph can be embedded as a directed graph using Algorithm 5.1. The transformation from sequence graph S to directed graph G is straightforward: each vertex or edge of S becomes a vertex or edge of G. However, this is a lossy transformation, since all of the information of the functions *start*, *end*, and *context* is ignored. Furthermore, there is no thread designation, as the edge sets of $\mathbf{E}$ are collapsed into a single edge set E of G. For convenience, we can define $|S| = |L \cup M| + \sum_{E_i \in \mathbf{E}} |E_i|$, and hence Algorithm 5.1 takes $O(|S|)$ time.

Input: Sequence graph $S = (L \cup M, \mathbf{E}, start, end, context)$
Output: Directed graph embedding of S

$V \leftarrow L \cup M$;
$E \leftarrow \emptyset$;
foreach $E_i \in \mathbf{E}$ **do**
    foreach $(u, v) \in E_i$ **do**
        $E \leftarrow E \cup (u, v)$;
    end
end
return *directed graph* $G = (V, E)$

Algorithm 5.1: EmbedSequenceGraph: Embeds a sequence graph as a directed graph.

When processing a sequence model, it is often necessary to identify all of the method activations according on a given lifeline. For example, drawing a sequence diagram requires extracting this information. Algorithm 5.2 presents a technique for extracting a sequence of method calls from a sequence graph. This approach takes $O(|M|)$ time, although in practice it can be done much faster. As the sequence model is built, the sequence of method activations for each lifeline can be maintained, which would allow the information to be available in constant time. The tradeoff is that $O(|M \cup L|)$ more space would be required, although processing each individual event would only require $O(1)$ more time. This has been deemed an acceptable design decision, and so JIVE does maintain redundant sequence graph information in order to optimize per-lifeline sequence extraction.

The implementation of the contour model for program execution follows from the description of its operational semantics as presented in Chapter 3. Method, static, and object contours are represented by classes that implement a `Contour` interface, which provides access to the contour's member table. Conceptually, a `Contour` is a *Composite* since an arbitrary contour can contain any number of nested contours [see *Composite* pattern in 32]. However, rather than distribute this structural information throughout the model, the topology of contours is maintained by a separate class, the `ContourModel`. An overview of the relationships between contours is shown in Figure 5.10.

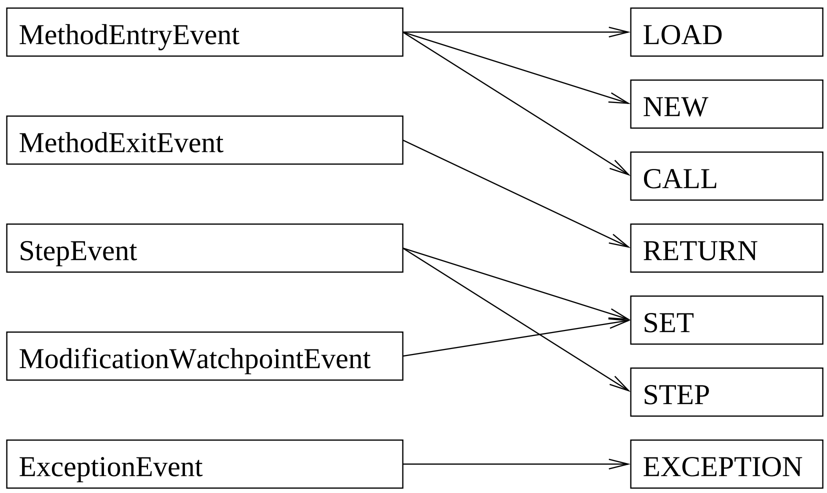


Figure 5.9: The relationships between JDI events and JiveLog Events. JDI events are listed on the left, and JiveLog events on the right. The arrows indicate which JDI events cause which JiveLog events.

Input: Sequence graph $S = (L \cup M, \mathbf{E}, start, end, context)$, Lifeline identifier $lifeline \in L$
Output: Sequence of intervals $[s_1, e_1]$, $[s_2, e_2]$, ..., $[s_n, e_n]$ representing the start and end times of the methods of $lifeline$

```
R ← ();
foreach m ∈ M do
    if context(m) = lifeline then
        append [start(m), end(m)] to R;
    end
end
return R
```

Algorithm 5.2: ExtractSequence: Extract a single lifeline's sequence of method intervals from a sequence graph.

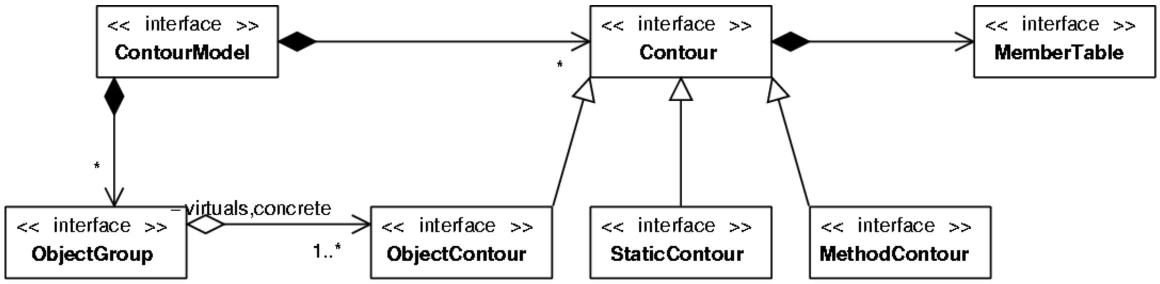


Figure 5.10: Relationships among contours.

It would be possible to distribute this topological data among the many contour objects, but this would make insertions and deletions rather complicated since the state of multiple objects would need to change synchronously. Centralizing the topological data also provides a single entrypoint for accessing contour information or modifying the contour model, and this simplifies the required behavior of listeners [see *Listener* pattern in 32].

A contour model is built from a subset of JiveLog events, as described below:

- *LOAD*: A LOAD event reflects a class' loading, and this is accompanied by the class' static contour being created in the static space of the contour model.

- *NEW*: A NEW event corresponds to the creation of an object group. In the contour model, an object contour for each context is created, and this new contour group is placed in the appropriate part of the model. Top-level objects are put in object space, and inner objects are placed within their enclosing object or static contour.

- *CALL*: When a method is called, a method contour must be created for it in the proper context of the contour model.

- *RETURN*: When a method returns, its method contour is removed from the contour model.

- *SET*: A SET event is fired when a variable changes. Within the contour model, the appropriate member table entry is updated with the new value.

- *EXCEPTION*: There are three distinct possibilities when an exception is thrown: the exception is caught within its throwing method, the exception is caught by another method, or the exception is uncaught. In the first case, nothing special is done with the contour model. In the second case, any methods that have abnormally exited due to the exception must be removed from the model. In the third case, all methods on the thread must be removed; additionally, there is a good chance that the program has terminated, and hence the contour model stops changing, leaving it as a snapshot of the state immediately prior to termination.

Each contour in a contour model has an associated contour identifier, a unique string that can be associated with the appropriate JiveLog context identifier; this translation is handled by the Mediator of Figure 5.1. Processing these references takes constant time, assuming a good hashing function. This means that the basic contour model operations of contour addition, contour removal, and variable value updates can be processed in constant time. Note that if a decentralized model was used instead, contour addition and removal would take $O(n)$ time, where n is the number of contours in the model, since adding a nested contour would require traversing through all of its ancestors.

Summary

JIVE is a large and complex piece of software, but most of its behavior can be broken down into the modules outlined at the beginning of this chapter. The true innovations in the JIVE architecture are in the online recording of program execution and the storage of these events in an execution database. Hence, the main contribution of the JIVE architecture is in the JiveLog module, although as a software artifact, it reifies our visual operational semantics for Java, and it serves as a platform for realizing our desiderata for interactive visualization. The next chapter explores the mathematical fundamentals of the interactive execution that is made possible through JIVE and JiveLog.

Chapter 6

Interactive and Reverse Execution

Debugging is an unavoidable aspect of software development, and yet common commercial debugging tools have not seen the significant advance present in other aspects of software engineering. There has been interesting and significant work done in classifying bugs, such as the works of Donald Knuth [60], Eric Allen [4], and Adam Barr [9], but this has not been integrated into many debugging tools. One of the primary difficulties of debugging is that, during normal execution, the evidence of a bug can only be detected after the bug's actual occurrence. In a traditional debugger, the process of finding a bug often involves guessing roughly where the bug is, setting a breakpoint, and running the program. Then, if one is lucky, the bug is found, but more often a breakpoint needs to be added or moved, or some debugging output must be produced, and the program is run again, often many times. Agrawal et al. [1] describe how backtracking, the capability to rewind execution, can assist in debugging. Many programmers readily accept this hypothesis, as quite often the capacity to step backward through program execution would allow detection of an error much faster and easier than setting a breakpoint and running the program again. In this chapter, we describe the JIVE methodology for *interactive execution*, the capacity to run a program forward or backward with the intention of explaining program semantics and therefore facilitating interactive debugging. The remainder of this chapter is structured as follows:

- Section 6.1 provides an overview and comparison of the two common approaches to interactive execution: *checkpointing* and *incremental state saving*. The JIVE methodology supports the latter.
- Section 6.2 describes a dynamic contour model for program execution, which facilitates the use of incremental state saving for efficient transitions between program states.
- Section 6.3 formally defines the set of *contour model transactions*, which are unit changes in a

```
public class MathUtils {
  public int factorial(int n){
    int i = 0;
    int result = 1;
    while (i < n) {
      result = result * i;
      i = i + 1;
    }
    return result;
  }
}
```

Listing 6.1: Java program that repeatedly changes a variable

dynamic contour model.

- Section 6.4 describes the software architecture that realizes the dynamic contour model and transaction database.

- Section 6.5 provides an example illustrating how the JIVE architecture and dynamic contour model can be used to efficiently and effectively represent a program's execution.

6.1 Comparison of Two Interactive Models

There are two general models for interactive execution using the state-saving model. One approach involves *checkpointing*, a technique whereby a copy is made of the program's execution state at a given execution point. This model allows for very efficient transitions between checkpoints; the time taken to restore the state is directly proportional to the size of the checkpoint. That is, the only time requirement is the time taken to eliminate old state information and restore the recorded state. The biggest drawback to this approach is that it takes a great deal of storage space, since the checkpoints are potentially quite large. Additionally, this space rises with inverse proportion to the frequency of checkpoints. Another drawback is that, in a pure checkpointing model, there is no way to directly restore states that occurred between checkpoints.

The other general approach to state-saving interactive execution is *incremental state saving*. This technique uses "checkpoints" as well, but in this model for interactive execution, only the changes between checkpoints are recorded, not the entire state. The amount of space required for this model is proportional to the size of the change between checkpoints. The affect of relative checkpoint distance depends on the operation of the program. For example, consider the program of Listing 6.1 that repeatedly changes variables' values. If a checkpoint is processed at each line of execution, then there will be `n` recorded changes to the variables `i` and `result`. On the other hand, if checkpoints are made at loop entrance and exit but not at each individual statement, then there would be only one recorded change for `i` and `result`. Specifically, at the end of the `while` loop, `i` and `result` would become n and $n!$, respectively. The space and time requirements therefore depend not only on the lexical distance between checkpoints, but also on the program's behavior with respect to its variables. When checkpoints are processed at every statement, there is a more direct correlation between the amount of space and time required and the running time of the program, since all modifications to variables are stored.

The full-state checkpointing approach has a benefit that any arbitrary (recorded) state can be restored very quickly. The time required to perform the restoration is essentially the time required to restore the checkpoint data. In this approach, the distance between checkpoints (that is, the number of intermediary recorded states) does not influence the processing requirements. The clear disadvantage is that there is a very high requirement on storage space. The space required for such a system grows linearly as the number of state recordings increases. Roughly, this number is linked to the running time of the program, although technically it depends on the number of times a monitored state-modifying statement is executed. By contrast, the incremental state saving model has the benefit that it requires relatively little storage space. At each transition, only the changes are stored, and so although the space requirement is still linearly proportional to the number of state-modifying statements executed, the constant is significantly lower. However, a transition to a disparate state can be much more computationally intensive than in the checkpointing model. In the incremental state saving model, in order to jump between disparate states, all intermediary states must be processed. When the total size of these state transitions exceeds the size of the state itself, this approach becomes less time-effective than full-state checkpointing. However, when the total size of transitions between states is less than the size of the program state, then it is possible for the incremental state saving model to be faster, although this strongly depends on the implementation. A hybrid approach involves taking intermittent, possibly incomplete checkpoints, and state transitions are recorded as well. Such an approach would achieve better time-effectiveness for the incremental state-saving approach, but it would also inherit the space requirements from both techniques.

It is interesting to note that the non-object-oriented precursor of JIVE used a full state-storage model for interactive execution combined with frequent, statement-level checkpointing. The intended application of this tool was students of programming languages, and the tool served to clarify the semantics of different parameter passing techniques using a contour model semantics similar to the one presented in Chapter 3. The programs used in this visualization tool were small programs intended to illustrate the applicability of the different parameter passing paradigms, and hence the state was not very large. The tool was able to be run on these examples without significant impact on the execution environment even though a simple in-memory data model was used. In fact, the only major performance bottleneck in the system occurred when using Java's printing functionality, which had only recently been integrated into the standard API.

6.2 Dynamic Contour Model

The formal contour model presented in Chapter 3 is used to define a specific state of execution. In order to model a dynamic execution environment, it is necessary to construct a *dynamic contour model*, a contour model that can formally change state. It is possible to make arbitrary changes to a contour model, but this undisciplined approach is difficult to analyze or quantify. Hence, we define specific types of changes that are legal, well-formed, and have clear and unique semantics. Within the JIVE methodology, the changes to a dynamic contour model reflect changes in a program's state. That is, the set of legal changes to the contour model corresponds to the set of legal changes in program state. Using the contour model semantics for Java as a basis, we can define three archetypical modifications that can be made to the dynamic contour model: the addition of a contour or contour group, the removal of a contour, and the changing of a value. These are explained in more detail in the following sections.

Adding a contour

In order for a contour addition transaction to be well-formed, the contour model must be complete before and after the addition. The semantics of this operation depend on the type of contour being added. Algorithms 6.1, 6.2, and 6.3 describe how each type of contour is added. For each algorithm, it is assumed that the input contour is already well-formed. This means, for example, that a method contour already has proper references to its thread of execution and its calling method contour. Adding a contour group is a matter of adding each object contour individually via Algorithm 6.2; note that the contours of an object group is already properly nested by definition.

Removing a contour

In order for a contour removal transaction to be well-formed, the contour model must be sound before and after the removal. This implies that there must be no references to the contour prior

Input: Contour model M,
static contour v to add to M,
optional enclosing contour $p \in M$
Output: Updated contour model

```
if p is specified then
    Add v to N(p);
    return M
else
    return ⟨v, O(M)⟩
end
```

Algorithm 6.1: AddStaticContour: Adds a static contour to the contour model.

Input: Contour model M,
object contour v to add to M,
optional enclosing contour $p \in M$
Output: Updated contour model

```
if p is specified then
    Add v to N(p);
else
    Add v to O(M);
end
return M
```

Algorithm 6.2: AddObjectContour: Adds an object contour to the contour model.

Input: Contour model M,
method contour m to add to M,
enclosing context $p \in M$
Output: Updated contour model

```
Add m to M(p);
return M
```

Algorithm 6.3: AddMethodContour: Adds a method contour to the contour model.

to removing the contour, since otherwise the model would invalidate the reference completeness property (Property 3.5). To remove any contour, it must be the case that the contour has no nested contours. Algorithms 6.4, 6.5, and 6.6 describe how the different types of contours are removed from the contour model. Object contour groups are removed by removing each member contour, one at a time, from the innermost to the outermost.

Input: Contour model M,
static contour $v \in M$ to remove
Output: Updated contour model

$p \leftarrow$ static contour in M that encloses v;
Remove v from $N(p)$;
return M

Algorithm 6.4: RemoveStaticContour: Remove a static contour from the contour model

Input: Contour model M,
object contour $v \in M$ to remove
Output: Updated contour model

if *v is a root contour* **then**
    Remove v from $O(M)$;
else
    $p \leftarrow$ contour in M that encloses v;
    Remove v from $N(p)$;
end
return M

Algorithm 6.5: RemoveObjectContour: Remove an object contour from a contour model.

Input: Contour model M,
method contour $m \in M$ to remove
Output: Updated contour model

$p \leftarrow$ contour in M that contains v;
Remove v from $N(p)$;
return M

Algorithm 6.6: RemoveMethodContour: Remove a method contour from a contour model.

Changing a variable's value

In order for a variable change transaction to be well-formed, the contour model must be complete before and after the update. The precise semantics of a variable's change in value depends on the type of the variable, the old value of the variable, and the new value being assigned. Algorithm 6.7 specifies how a variable's value is changed with respect to the contour model. Keep in mind that, in Java, there is a pointed difference between primitive and reference types.

Input: Contour model M,
the variable (or field) f that is changing,
the contour $v \in M$ such that $f \in E(v)$,
the new value x of f
Output: Updated contour model

if *type*(f) *is not primitive* **then**
  if *value*$(f) \neq$ *null* **then**
    Remove *value*(f) from $A(v)$;
  end
  if $x \neq$ *null* **then**
    Add the contour corresponding to x to $A(v)$;
  end
end
Set *value*(f) to x;
return M

Algorithm 6.7: ChangeValue: Change the value of a variable in the contour model

6.3 Transactions

Section 3.5 presented a mathematical representation for static (*i.e.* non-dynamic) contour models. A dynamic contour model can be said to have a state s such that s is a sound static contour model. That is, at a stable state, a dynamic contour model has the same representation as a static contour model. The modifications described in the previous section can then be interpreted as transitions between states, contour model "deltas." We refer to these as *transactions* on the contour model, and an arbitrary transaction is represented as δ. Further borrowing from database nomenclature, *committing* a transaction means applying it to a dynamic contour model in order to produce a new state; given a contour model in state s_i and a transaction δ, we denote the commit as $s_i + \delta$, which produces a new state s_{i+1}. By convention, we let s_0 represent an empty contour model; therefore, the given a dynamic contour model that has had n transactions committed on it, we can express the history of dynamic contour model states as a sequence $s_0, s_1, \ldots, s_n$ where the sequence of transactions is $\delta_1, \delta_2, \ldots, \delta_n$, such that $s_i + \delta_{i+1} = s_{i+1}$ for all $1 \leq i < n$.

A dynamic contour model can also be reverted to previous states. Given a dynamic contour model in a state s_i, where $i > 0$ holds and δ_i is the transaction that satisfies $s_{i-1} + \delta_i = s_i$, then we can *roll back* the transaction δ_i on the contour model in order to revert it to state s_{i-1}. We denote this rolling back operation as $s_i - \delta_i = s_{i-1}$.

Transaction Definitions

The specific transactions that we define over the contour model are based directly on the dynamic contour model operations presented earlier in this chapter, and they are defined below:

- **AddContour:** This transaction is used to add a static or method contour to a contour model. Its rollback operation is to remove this contour from the model.

- **AddContourGroup:** This transaction is used to add an object group to the contour model. An object group is composed of n object contours, a "stack" of $n-1$ virtual object contours with one deeply nested concrete object contour. This event is used to add trivial contour groups of size one; these only occur within Java when an instance of `java.lang.Object` is created directly. Contours in a contour group are added at once from the perspective of the dynamic contour model and the transaction model; they are not a series of n AddContour transactions. The rollback operation for this transaction is the removal of the contour group.

- **RemoveContour:** This transaction is used to remove a method contour from the contour model. Static contours are not removed from a contour model during forward execution, only methods are, upon their completion or exceptional exiting. The rollback operation for this transaction is the re-addition of the contour.

- **RemoveContourGroup:** This transaction is used to remove a contour group from the contour model. Since a contor group represents a single Java object, and the current JIVE architecture does not monitor object destruction or reclamation by the garbage collector, this transaction type is not ccurrentlyused. It is included in the definitions of transactions for theoretical completeness, since this operation is technically possible on the contour model as a pair to AddContourGroup. The rollback operation for this transaction is the re-addition of the contour group.

- **ChangeValue:** This transaction is used to change the value of a variable within the member table of a contour in the contour model. The rollback operation of this transaction is the setting of the variable to its previous value.

We also define a *step transaction*, which is a transaction that is the last transaction of its step in the forward direction. This model assumes the definition of a step as the execution of a line of course code, which is roughly equivalent to a statement. Obvious exceptions are loops or conditionals: a loop or conditional whose condition is on a separate line from its body will have its condition tested as a separate logical step, whereas one that is on the same line will generate only one step. Step transactions are annotated with an asterisk in the transaction sequence. For example, a transaction δ_i that is a step transaction would be written as δ_i^* wherever the annotation of step transactions is appropriate. A *program step* or *logical program step* is defined by the series of transactions $\delta_i, \delta_{i+1}, \ldots, \delta_j$ such that:

- the transaction δ_{i-1} is a step transaction or it does not exist (*i.e.* $i = 0$).
- the transaction δ_j is a step transaction or it is the last transaction in the transaction sequence.

Model Soundness

As mentioned previously, arbitrary modifications to the dynamic contour model cannot be made. For example, consider a dynamic contour model with two states, s_0 and s_1, and let δ_1 be the transaction such that $s_0 + \delta_1 = s_1$. Furthermore, let δ_1 be an AddContour transaction that adds a static contour `java.lang.Object` to the model, where the static contour represents the static environment of the class of the same name. Let δ^* be a ChangeValue transaction that updates the value of a variable `v` within a contour `C:3` to the value 10. Then it makes no sense to compute a state s^* where $s^* = s_1 - \delta^*$; the transaction cannot be rolled back on a dynamic contour model in state s_1 since there is no contour `C:3`, let alone a variable `v`.

This example illustrates the fact that a restriction is necessary on the transactions that are committed or rolled back on a dynamic contour model. Given a ddynamiccontour model M that has a state history $s_0, s_1, \ldots, s_n$, we define that sequence of states as the `state sequence` of M. The list of transactions $\delta_1, \delta_2, \ldots, \delta_n$ where $s_i + \delta_{i+1} = s_{i+1}$ is defined as the *transaction sequence* of M. Then we assert that, by definition, the only legal rollback from any state s_j in the transaction sequence of M, where $0 < j \leq n$, is the rollback of transaction δ_j such that $s_j - \delta_j = s_{j-1}$.

The previous paragraph defines the soundness of a dynamic contour model with respect to rolling back. Soundness with respect to committed transactions is satisfied as long as the following criteria are met for a dynamic contour model M whose state sequence is $s_0, s_1, \ldots, s_n$ and whose transaction sequence is $\delta_0, \delta_1, \ldots, \delta_n$:

- For each AddContour transaction δ_i committed on a dynamic contour model in state s_i, if the contour is added within an enclosing context C, then C is in the contour model s_i.
- For each AddContourGroup transaction δ_i committed on a dynamic contour model in state s_i, if the contour group is added within an enclosing context C, then C is in the contour model

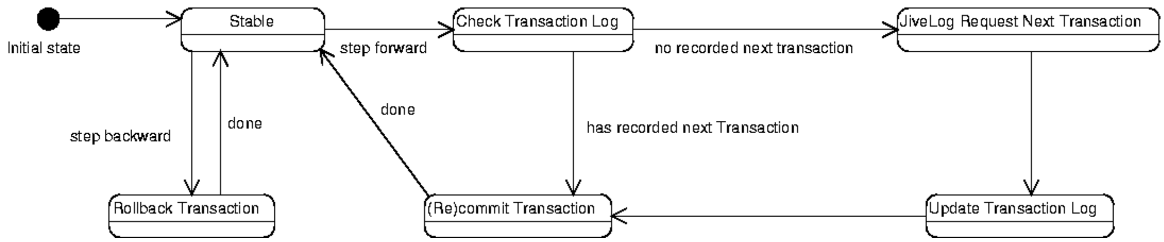


Figure 6.1: State diagram for interactive execution with a dynamic contour model

s_i.

- For each RemoveContour transaction δ_i committed on a dynamic contour model in state s_i, the contour being removed must be in s_i.
- For each RemoveContourGroup transaction δ_i committed on a dynamic contour model in state s_i, the contour group being removed must be in s_i.
- For each ChangeValue transaction δ_i committed on a dynamic contour model in state s_i, the variable being changed must be in the contour model s_i. If the new value for the variable is a reference to a contour C, then C must also be in s_i.

If all of the commits of a transaction sequence satisfy these criteria and the criteria for rollback soundness, then the dynamic contour model is said to be sound. Figure 6.1 presents a UML state diagram illustrating the stable states in processing a dynamic contour model.

6.4 Architecture for Interactive Execution

The dynamic contour model is a machine that represents the changing states of a Java program. Our fourth desideratum for interactive program execution is for the support of forward- and reverse-execution of programs (see Section 1.2). The contour model provides a unique means for enabling such interactive execution. In a sense, it provides a visual mechanism for computation, although its computations are limited to contour-centric operations. Forward execution, then, is a matter of committing transactions to the dynamic contour model, and reverse execution is rolling them back, and as long as the model remains sound, a representation of program state is provided through the visual operational semantics for Java.

Modifications to the contour model are embodied as `Transaction` objects. The `Transaction` interface defines two methods: `commit(ContourModel)` and `rollback(ContourModel)`. A `Transaction` is a *command* in the Command pattern, and parameterizing the contour model (making extrinsic

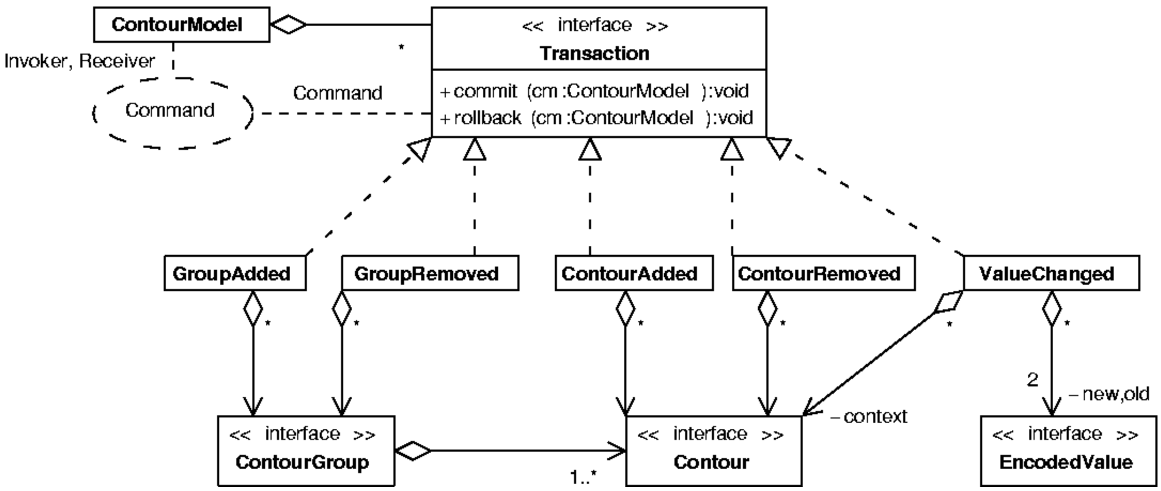


Figure 6.2: A UML class diagram representing the contour model transaction module design. The *ConcreteCommand* participants in the *Command* collaboration have not been shown in order to reduce the crossing of dashed edges [see *Command* pattern in 32]; the participants are the concrete iimplementersof `Transaction`.

this attribute of the transaction) facilitates using flyweight transaction objects [see *Command* and *Flyweight* patterns in 32]. Using nomenclature derived from the perspective of the contour model as a database of execution state information, a transaction can be committed or rolled-back on the contour model. The specific types of transactions are: adding or removing an individual static or method contour; adding or removing an object group; and modification to a value. Each transaction object takes approximately 16 bytes of memory; this is an amortized analysis and does not consider overhead imposed by the JVM, as described by Sosnoski [84]. The space used by this model is $O(t)$, where t is the number of changes made to the contour model during the program's execution (roughly equivalent to the duration of the program). Figure 6.2 outlines the design of the transactions module.

JiveLog Integration

In order to roll back transactions, they must necessarily have been recorded already. Committing prerecorded transactions, as occurs after reverse execution, is likewise simple since the transaction is already within the log. During forward execution, if there are no recorded states (*i.e.* the current state is s_k and the last recorded transaction is δ_k), then new transactions must be obtained. In the JIVE methodology, these transactions are interpreted from JiveLog events. JiveLog is the mechanism and format in which Java program execution is recorded by JIVE and was introduced in Chapter 5.

JiveLog events are translated into transactions using the mapping shown in Figure 6.3. The

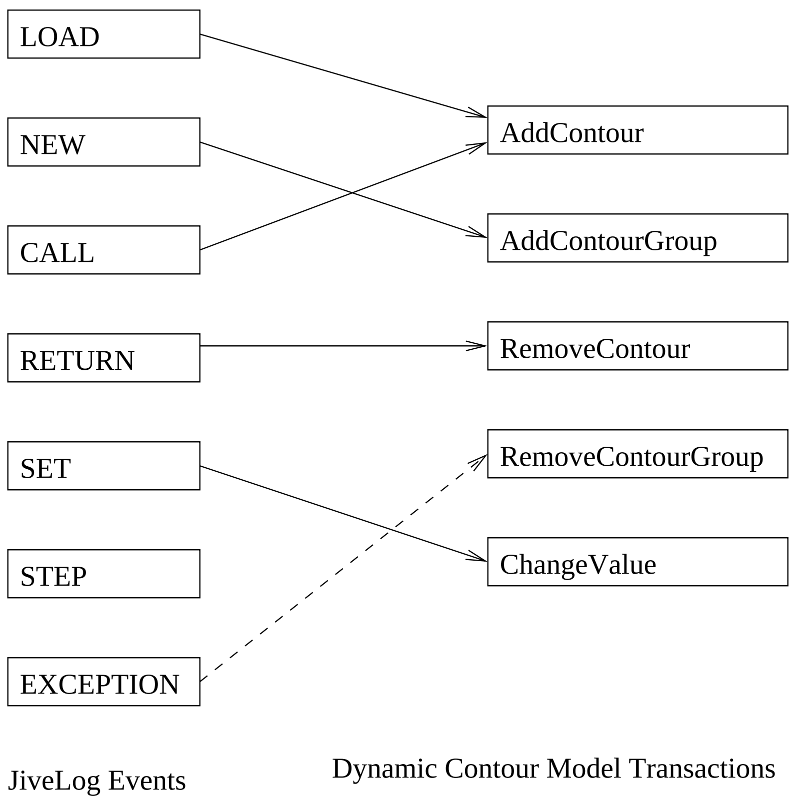


Figure 6.3: The translation of JiveLog events to dynamic contour model transactions. The solid lines indicate definite translation, and the dashed line indicates conditional translation: EXCEPTION events are only interpreted as RemoveContour transactions under specific circumstances.

translations are further described below:

- *LOAD* events, corresponding to class preparation and therefore the initialization of the static context, are interpreted into AddContour transactions for the class' static contour.
- *NEW* events, corresponding to object creation, are interpreted into AddContourGroup transactions, where the contour group is created for the new object group.
- *CALL* events, corresponding to method invocation, are interpreted into AddContour transactions for the method's contour. A CALL is always interpreted as a step transaction.
- *RETURN* events, corresponding to method return, are interpreted into RemoveContour transactions for the method's contour. A RETURN is always interpreted as a step transaction.
- *SET* events, corresponding to a change in variable's value, are interpreted into ChangeValue transactions.
- *EXCEPTION* events, corresponding to an exception's being thrown, are interpreted into different transaction sets depending on the nature of the exception. If the exception is thrown and caught in the same method, no transaction is generated since there is no dynamic contour model modification. If the exception is caught in a context different from its thrower, then RemoveContour transactions are generated for the contour of each method that is exited. An EXCEPTION event is always interpreted as a step transaction.

An algorithm to compute this transformation is presented as Algorithm 6.8. The algorithm specifies how step transactions are determined.

Interactive Controls

We define five interactive controls for the JIVE front-end to the dynamic contour model representation of program execution. The visual interface to these controls are shown as the leftmost five buttons of Figure 6.4, a screenshot of JIVE's main toolbar. These controls are described below.

Run backward (Rewind) : Runs the program backward to the initial state or the first breakpoint encountered.

Step backward : Run the program backward one program step.

Interrupt (Pause) : Interrupt either run backward or run forward mode and pause execution.

Input: JiveLog Event Sequence $S = e_1, e_2, \ldots, e_n$
Output: Dynamic contour model transaction sequence

$j \leftarrow 1$;
$T \leftarrow$ empty transaction sequence;
foreach *Event* e_i *in* S, $i = 1, 2, \ldots, n$ **do**
    if e_i *is a LOAD event* **then**
        $\delta_j \leftarrow$ new AddContour transaction;
        Add δ_j to T and increment j;
    end
    if e_i *is a NEW event* **then**
        $\delta_j \leftarrow$ new AddContourGroup transaction;
        foreach *virtual context* c *created in* e_i **do**
            Add a virtual object contour for c to the contour group of δ_j;
        end
        Add a concrete object contour to δ_j for the concrete object of e_i;
        Add δ_j to T and increment j;
    end
    if e_i *is a CALL event* **then**
        $\delta_j^* \leftarrow$ new AddContour step transaction;
        Add δ_j^* to T and increment j;
    end
    if e_i *is a RETURN event* **then**
        $\delta_j^* \leftarrow$ new RemoveContour step transaction;
        Add δ_j^* to T and increment j;
    end
    if e_i *is a SET event* **then**
        $\delta_j \leftarrow$ new ChangeValue transaction;
        Add δ_j to T and increment j;
    end
    if e_i *is a STEP event* **then**
        if δ_{j-2} *is not a step transaction* **then**
            Replace δ_{j-2} in T by an equivalent a step transaction δ_{j-2}^*;
        end
    end
    if e_i *is an EXCEPTION event* **then**
        foreach *method* m *exited by* e_i **do**
            if m *is the last method exited by* e_i **then**
                $\delta_j^* \leftarrow$ new RemoveContour step transaction;
                Add δ_j^* to T and increment j;
            else
                $\delta_j \leftarrow$ new RemoveContour transaction;
                Add δ_j to T and increment j;
            end
        end
    end
end
return T

Algorithm 6.8: EventToTransaction: Translates JiveLog events into dynamic contour model transactions.

Figure 6.4: A screenshot of the JIVE toolbar. From left to right, these controls are run backward, step backward, pause, step forward, run forward, show source code, stacked view diagram, minimized view diagram, detail view diagram, and call-path diagram.

Step forward : Run the program forward one program step.

Run forward (Fast-Forward) : Run the program forward to its termination or to the first breakpoint encountered.

Breakpoints are defined by source code locations, as in a traditional debugger. However, the context of interactive execution in JIVE gives breakpoints a novel functionality: they can be used to stop reverse execution as well as forward execution. Within the JIVE architecture, the reaching of breakpoints is detected by monitoring CALL, STEP, and EXCEPTION events, each of which incorporate a change in line number. Breakpoints are set and removed by clicking on the margin of the appropriate line in JIVE's source code view pane. If B is the set of breakpoints, then this technique does impart $O(|B|)$ additional time to the processing of each CALL, STEP, and EXCEPTION event. However, in practice $|B|$ is quite small, and so this does not realistically impact the performance of the system.

Observations

The first observation to make regarding this approach is that the client program is not altered. There are systems that actually reverse the state of the Java Virtual Machine, such as the work of [23]. However, such an approach necessarily invalidates our seventh desiderata for interactive program visualization since it uses a custom Java Virtual Machine, and therefore off-the-shelf Java technologies cannot be used to run the visualization. A corollary of this observation is that input cannot be re-entered or modified during an execution with JIVE. Once any resource is read that is not strictly program-controlled, such as user input, it is inexorably stored as a transaction and cannot be modified in the future. On one hand, this is a shortcoming of the system, since one cannot alter a program's input during interactive execution, a functionality that doubtlessly would be useful for debugging. On the other hand, this approach has a strong advantage in that it totally avoids the viscosity problem. This problem arises when states are recorded, then rewound, and then an value (such as user input) is altered. All recorded states that follow the current state are potentially

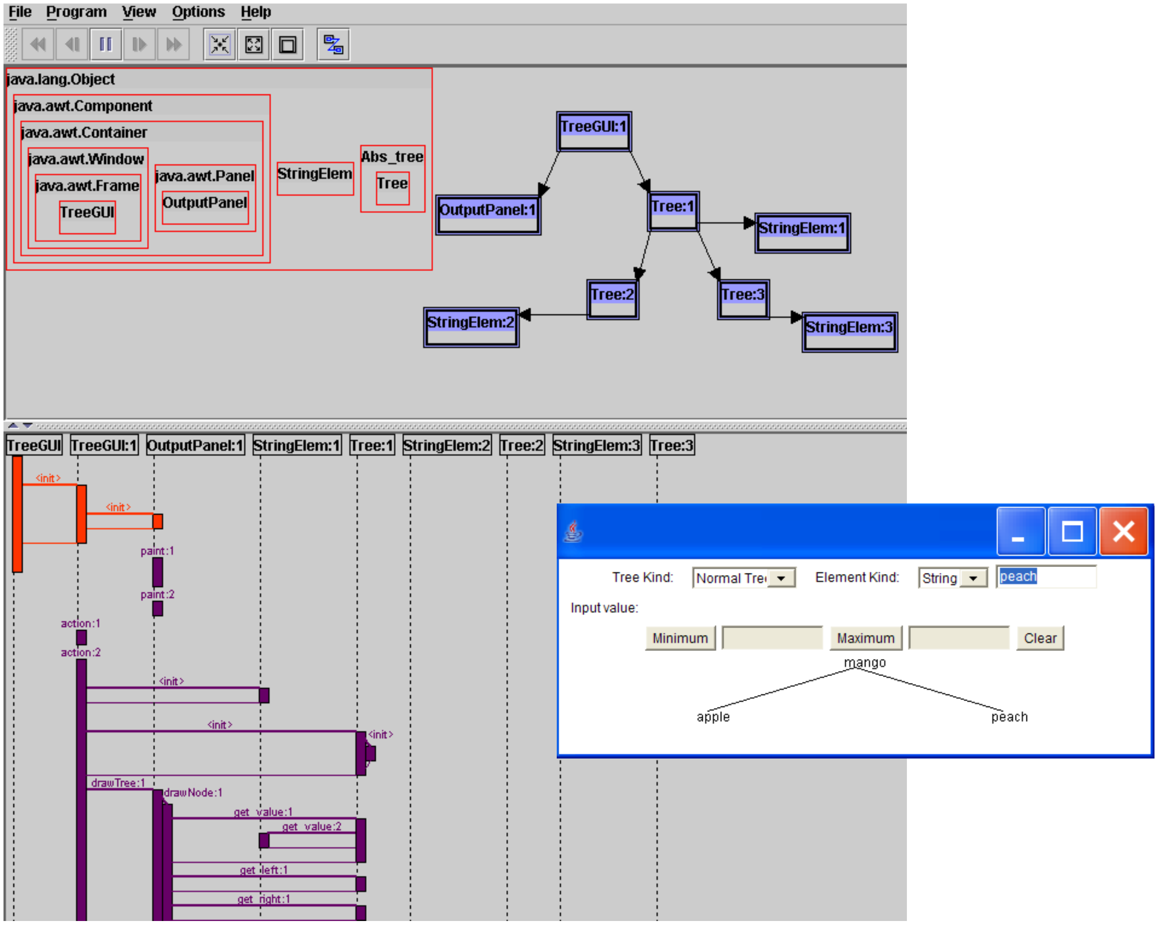


Figure 6.5: A screenshot of JIVE and the GUI of the program it is visualizing. The client program is an application that constructs and draws binary search trees of strings.

invalidated, since they might depend on the altered value. The viscosity problem is difficult to solve, and it would be impossible to address within the JIVE methodology since, as mentioned above, we do not send instructions directly to the client VM; it is therefore impossible request re-computations of defunct methods on altered input.

The second observation to make regarding this approach to interactive execution is that it facilitates the visualization of programs that have their own graphical user interfaces (GUIs). Figure 6.5 provides a screenshot of JIVE being run alongside the GUI of the program it is currently visualizing. This is possible because the client program is not modified. If an attempt were made to reverse-execute the client, it would also need to reverse-update its own GUI; this is a difficult if not impossible proposition.

The third observation to make on this approach to interactive execution is that it automatically and seamlessly handles multithreaded applications. Each transaction object can be annotated with

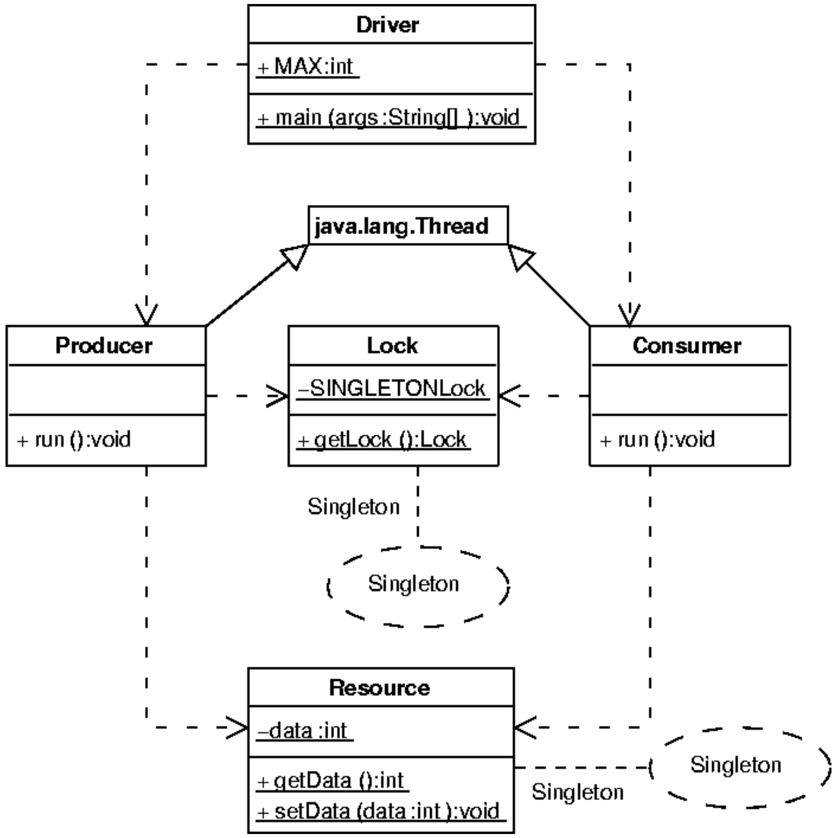


Figure 6.6: Class diagram for the Producer/Consumer example

the thread on which the transaction has occurred. Thread scheduling is managed by the client application's Java Virtual Machine, and the scheduling is not affected by JIVE or its JPDA-based implementation. When reverse executing, transactions are necessarily rolled back in the opposite order in which they were committed, which by design does not take threads into account. Therefore, the order of thread execution is preserved within the dynamic contour model in its transaction log. Additionally, repeated forward and reverse execution of the same execution log will reconstruct exactly the original execution; if threads were rescheduled during re-execution, then it is likely that multithreaded programs would have their threads scheduled in a different order, which could lead to different execution logs in the best case and different synchronization errors in the worst.

6.5 Example

In this section, we provide an example of how JIVE's interactive execution architecture works on a simple program. The example we will consider is a multithreaded producer/consumer application whose architecture is shown in Figure 6.6. The `Producer` and `Consumer` classes both subclass

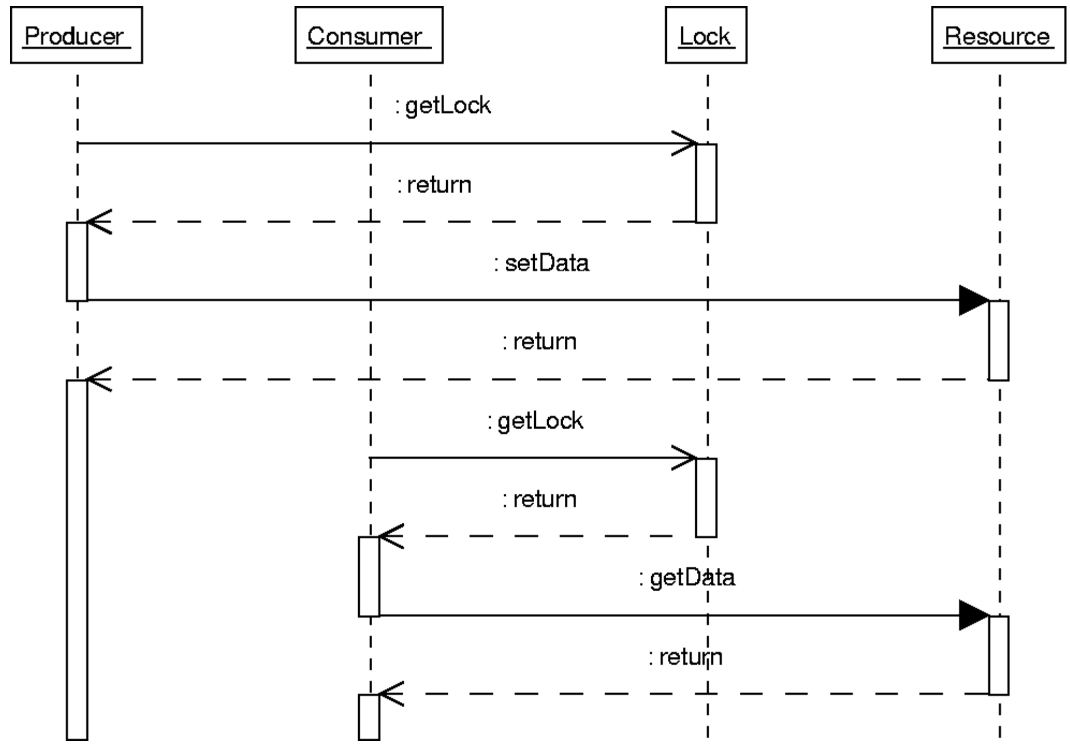


Figure 6.7: Sequence diagram for the Producer/Consumer example. This sequence diagram presents the ideal interaction of objects, where the `Producer` thread is scheduled to run before the `Consumer`.

`java.lang.Thread`; this is the means by which threads of execution are created in Java's standard API. The `Lock` class provides a singleton lock object on which synchronization must be acquired prior to accessing the static `Resource` class. The `Resource` class is also a singleton of sorts, though not in the traditional sense presented in Gamma et al. [32]: it exhibits the same behavior as a singleton by keeping its state and methods `static`. The function of the `Resource` class is to hold a single integer. The integer value is provided by the `Producer` and extracted by the `Consumer`. Figure 6.7 provides a sequence diagram illustrating the ideal interaction situation where the `Producer` sets the `Resource` value prior to the `Consumer`'s accessing it. This simple application does not provide rigorous error-checking or synchronization testing, but it illustrates the behavior of the interactive execution model described in this chapter.

The `main` method of the `Driver` class from the Producer/Consumer example is shown in Listing 6.2. This is the first method to be executed in the program, and so its class must be loaded first. Keep in mind that in the JIVE methodology, we only track a relatively small subset of the actual classes; this is usually the set of classes for which the source is available, as in this example. Hence, we ignore classes such as the `ClassLoader` and `Runtime`, both of which are integral to the Java Virtual Machine initialization and the launching of applications. The following series of steps illustrates

```
public class Driver {
  public static void main(String[] args) {
    Producer p = new Producer();
    Consumer c = new Consumer();
    p.start();
    c.start();
  }
}
```

Listing 6.2: Driver for the Producer/Consumer example

Event Number	**JiveLog Event**	**Transaction**
1	LOAD the `Object` class	AddContour: static `Object` contour
2	LOAD the `Driver` class	AddContour: static `Driver` contour within `Object`
3	CALL the `main:1` method	AddContour: `main:1` method within `Driver`
4	CREATE the `Producer` object	AddObjectGroup: `Object:1`, `Thread:1`, and `Producer:1` group
5	SET the value of `p`	ChangeValue: set `p` in `main:1` to `Producer:1`
6	STEP to line 5	*none*
7	CREATE the `Consumer` object	AddObjectGroup: `Object:2`, `Thread:2`, and `Consumer:1` group
8	SET the value of `c`	ChangeValue: set `c` in `main:1` to `Consumer:1`
9	STEP to line 6	*none*

Table 6.1: First 9 events of the Producer/Consumer example execution

the first few JiveLog events that JIVE generates for the execution of the program along with the associated dynamic contour model transactions; the events are also summarized in Table 6.1.

1. The first event that is recorded by JiveLog is the loading of the `java.lang.Object` class. Strictly speaking, this class should be exempted from visualization since we do not have access to its source-code. However, a special exception is made for the `Object` class, which serves two important roles in the contour model semantics for Java: first, its static contour is necessarily the ultimate superclass of all other static contours, and second, an `Object` instance contour is the outermost virtual contour for all object groups except those that are formed solely by a single concrete `Object` instance contour. This event is translated into a dynamic contour model AddContour event, which adds a contour `java.lang.Object` to static space.

2. The `Driver` class is loaded, as reflected by the second LOAD event. This is translated into the addition of the static `Driver` contour to the contour model. This contour is nested within `java.lang.Object`, which represents its superclass' static context, in static space.

Transaction	Transaction Type	JiveLog Event Number
δ_1	AddContour	1
δ_2	AddContour	2
δ_3	AddContour	3
δ_4	AddContourGroup	4
δ_5	ChangeValue	5
δ_6	AddContourGroup	7
δ_7	ChangeValue	8

Table 6.2: Transactions for the Producer/Consumer example. The JiveLog event numbers refer to the numbers in Table 6.1. Events 6 and 9 do not generate transactions.

3. With `java.lang.Object` and `Driver` loaded, the `main` method is executed, as represented in JiveLog by a CALL event. This is interpreted as another AddContour transaction on the contour model, with the contour being a new `main:1` method contour, which is nested within the static `Driver` contour.

4. The creation of a `Producer` object via the `new` operator is the next significant operation, and it is represented in the JiveLog CREATE event. The object group contains three contours: a virtual `Object` contour, `Object:1`; a virtual `Thread` contour, `Thread:1`, which is nested within `Object:1`; and a concrete `Producer` contour, `Producer:1`, which is nested within `Thread:1`. This contour group is the first added to object space during the execution of the program. This JiveLog event is interpreted as an AddContourGroup transaction involving the contour group described above.

5. After the `Producer` object is created, a reference to it is assigned to the variable `p`; this is captured by JiveLog in a SET event. There is no constructor in `Producer`, and so any implicit invocation of superclass constructors is hidden by the default JiveLog event filter. In the contour model, this is interpreted as a ChangeValue event on `p` in `main:1`.

6. STEP events are used in JiveLog to indicate the ends of statements. This is required for step-wise interactive execution, although there is no explicit modification to the dynamic contour model. Instead, a marker can be placed in the transaction sequence of the contour model indicating that the previous transaction was the last on its step. In this case, the previous AddContourGroup and ChangeValue transactions would be considered to be one logical step.

7–9. These events are processed very similarly to the previous three, with the differences highlighted by the summary of Table 6.1.

The transactions described above are labeled δ_1 through δ_7, as illustrated in Table 6.2. Figure 6.8

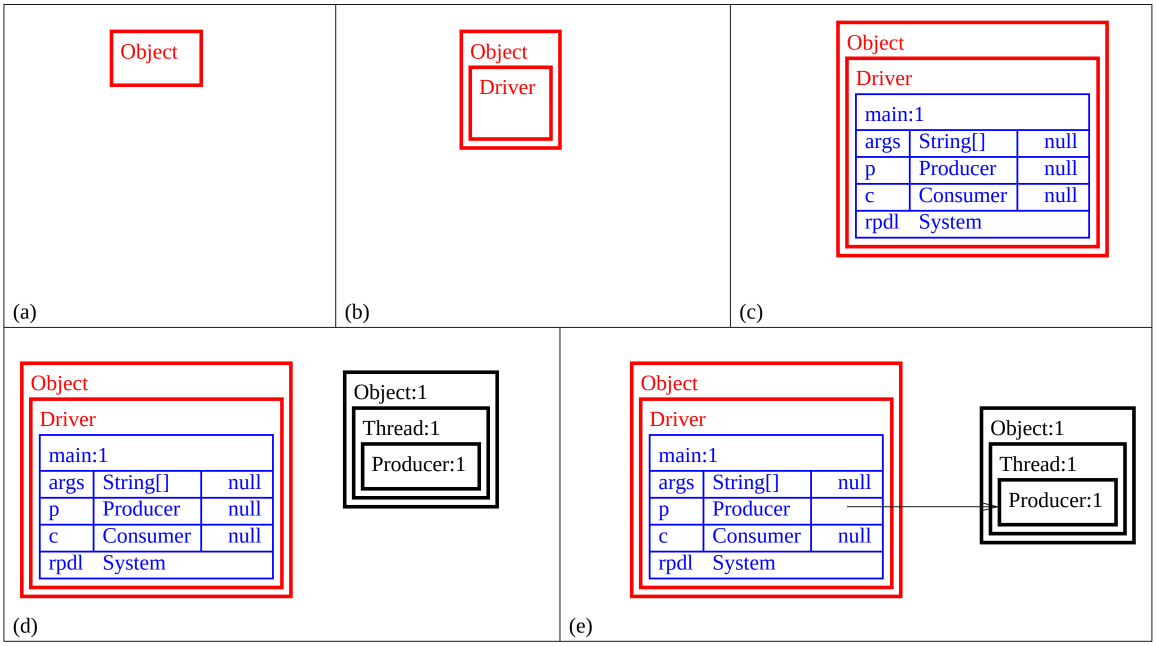


Figure 6.8: First five states of the dynamic contour model in the Producer/Consumer example. The initial state s_0, which is the empty contour model, is not shown.

presents a series of contour diagrams illustrating some of the states of the dynamic contour model during the execution sequence. Let us consider the state shown in Figure 6.8(e), which is state s_5, produced after the processing of JiveLog event number 5, the assigning of `Producer:1` to `p` in `main:1`. Revers executing from this state requires the rolling back of transaction δ_5, a ChangeValue transaction. By our definition, such a transaction stores the new and old values of the target variable. Hence, rolling back the transaction can easily revert `p` in `main:1` to `null`. This is a transition from s_5 to s_4, which is equivalent to a change from (e) to (d) in Figure 6.8. To return to s_5 involves a re-commit of transaction δ_5, the ChangeValue transaction, which recorded that the new value of `p` would be `Producer:1`. Stepping forward again then will recommit δ_5, returning the contour model to the state illustrated in Figure 6.8(e).

Once both threads are initialized and running, the thread on which JiveLog events are recorded becomes important to the dynamic contour model transactions. Both the `Producer` and `Consumer` threads use the `Lock` to synchronize access to the `Resource` singleton, as illustrated in Listings 6.3 and 6.4. Once these threads are running, a state such as that of Figure 6.9 can be obtained. The `Lock` instance would have been created during the static initialization of the `Lock` class. This would occur after the class is loaded, which is after the first time it is referenced. Hence, the `Object:3` contour's number is significant since `Lock:1` isn't actually created until after `Producer:1` and `Consumer:1`.

```java
class Producer extends Thread {
  public void run() {
    for (int i=1; i<=Driver.MAX; i++) {
      synchronized (Lock.getLock()) {
        if (Resource.getData()==0) Resource.setData(i);
      }
      try {
        sleep(1);
      } catch (InterruptedException e) {
      }
    }
  }
}
```

Listing 6.3: Producer implementation

```java
class Consumer extends Thread {
  public void run() {
    while (true) {
      synchronized (Lock.getLock()) {
        if (Resource.getData()!=0) {
          System.out.println("Read: " + Resource.getData());
          if (Resource.getData()==ProducerConsumer.MAX)
            System.exit(0);
          else Resource.setData(0);
        }
      }
      try {
        sleep(1);
      } catch (InterruptedException ie) {
      }
    }
  }
}
```

Listing 6.4: Consumer implementation

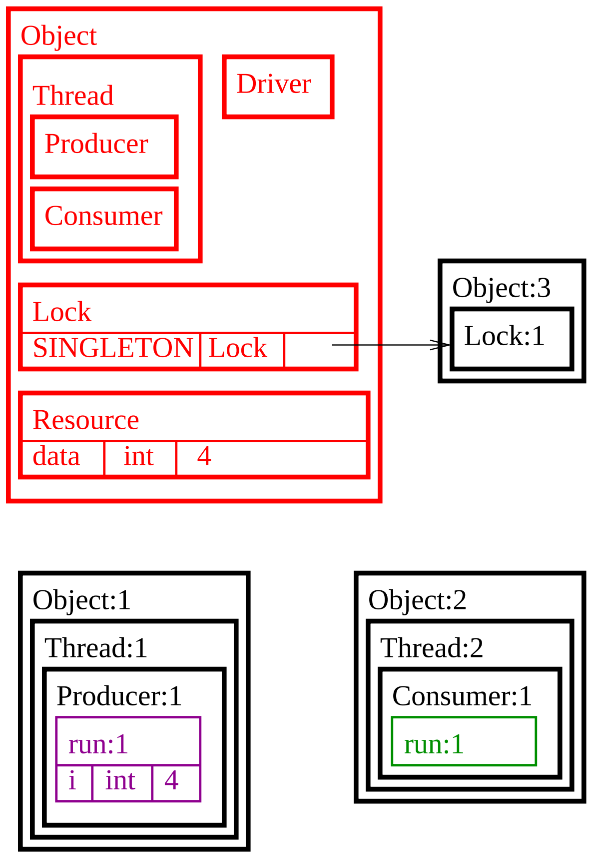


Figure 6.9: A sample state of the Producer/Consumer example. The original thread, which ran `main`, has terminated. The two new threads of execution are shown in purple and green in this diagram.

Chapter 7

Drawing Object Diagrams

JIVE's enhanced object diagrams pose significant challenges to automatic drawing. There is a great deal of research on drawing general graphs (see Chapter 2), but our object diagrams cannot be directly treated as graphs for several reasons: nodes contain nested structures, there are multiple types of edges, and there are meaningful crossings that should not be eliminated. Therefore, custom techniques are required to process these object diagrams and produce good automatic drawing.

Automatic graph drawing is essentially a problem of satisfying conflicting aesthetic constraints, and it is therefore combinatorially hard. In this chapter, we present a precise specification of the aesthetic preferences and drawing constraints for enhanced object diagrams. We show that by careful processing of the object diagram, it can be converted into a graph such that existing techniques can be applied with minor modifications. The remainder of this chapter is structured as follows:

- Section 7.1 describes the aesthetic preferences and drawing constraints for automatic drawing of enhanced object diagrams.
- Section 7.2 presents our algorithm for converting an object diagram to a directed multigraph.
- Section 7.3 presents our layered hierarchical drawing algorithms for object diagrams.

7.1 Aesthetic Preferences

Graph drawing research is driven by the specification of drawing conventions, aesthetic preferences, constraints, and efficiency [24]. In this section, we examine the aesthetic preferences of drawing enhanced object diagrams and its effect on our selection of drawing algorithms. This is followed by a discussion of the conventions apply onto our drawing methodology. As mentioned previously, a contour model is not a pure mathematical graph, and hence a contour diagram cannot necessarily be treated as a drawing of a graph. If this were the case, then drawing object diagrams would be a simple matter of applying known techniques such as the `dot` algorithm [33] to produce good drawings of the graph. The nature of the contour model complicate the application of traditional graph drawing aesthetics. The specific properties of the contour model that must be addressed are its heterogeneous contours, meaningful crossings, nested structures, and heterogeneous edges.

Heterogeneous Contours

The contour model contains three types of contours: static contours, object contours, and method contours. Although the precise semantics of any contour depend on the program unit it represents, the different types of contours share basic behaviors. Specifically, the lifecycle of each is significantly different:

- Static contours exist in static space and represent a Java Virtual Machine's *class objects*. They come into being when a class is loaded, and they are never removed or destroyed since Java does not define a class unloading operation.

- Object contours represent instances of classes, and hence they are created during object creation. Java objects cannot be explicitly destroyed; they exist until reclaimed by the garbage collector. Hence, object contours have a semipermanent status. Practical considerations prohibit monitoring object reclamation: to monitor an object requires a reference to it, and having a reference to it prevents reclamation. Furthermore, the JPDA does not provide object destruction notification, and hence our JPDA-based JIVE architecture cannot be used to monitor objects for destruction. Object contours therefore have approximately the same lifecycle as static contours, albeit due to practical and not theoretical reasons.

- Method contours represent method activations and must exist within static or object contours. Method activations have distinct call and return points in a program's execution sequence, and therefore method contours have distinct points of creation (addition to the model) and destruction (removal from the model).

The above observations support the following property of the contour model:

Property 7.8: Method Transience. Method contours are the only type of contour that are necessarily removed from the contour model during program execution.

This property has important implications for the behavior of structural links. Structural links from static or object contours generally share the lifecycle of their source contour, modulo changes

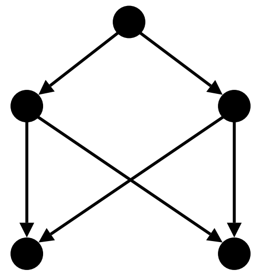


Figure 7.1: Necessary crossings in a layered drawing. This graph cannot be drawn without crossings if the diagram is to be upward, layered, and containing only straight lines.

in the corresponding variable's value. Similarly, structural links from method contours share the lifecycle of the method (modulo explicit value modification), implying that they generally are transient. This generalization is clearly not a law of programs since it is trivial to create a program where reference fields fluctuate in value more often than method variables; however, regardless of the behavior of object fields, an important corollary of Property 7.1 is that method variables are necessarily destroyed when their corresponding method contour is destroyed.

Meaningful Crossings

A well-established aesthetic property of good graph drawings is that they have no or few edge crossings [see *e.g.* 24]. Some graphs cannot have all crossings removed due to their structure and drawing constraints. For example, Figure 7.1 presents a simple graph where the crossings cannot be removed if the graph is drawn upward and using only straight lines. The crossing in the figure is an artifact of the graph's structure and has no inherent or structural meaning. In fact, all crossings can be removed in a straight-line graph drawing if the drawing space is three-dimensional. However, this approach assumes that crossings, as in Figure 7.1, are meaningless artifacts of the graph's drawing. This is not the case in the contour model, where some types of crossings may contain important information that should not be removed from the diagram.

First, we observe that there are three types of crossings in an enhanced object diagram:

Definition 7.1. An **edge-edge crossing** is a crossing between two links, which are generally rendered as arcs in a traditional graph drawing. Figure 7.1 illustrates an edge-edge crossing. Edge-edge crossings are never meaningful; they are always artifacts of a contour diagram rendering.

Definition 7.2. An **edge-border crossing** is a crossing between a link (edge) and a contour border. Figure 7.2 illustrates several edge-border crossings with a single structural link. Edge-border crossings can be meaningful or meaningless. In Figure 7.2, all the edge-border crossings are necessary

```
public class LinkedList {
  private class Node {
    Node next;
    Object data;
  }
  private Node head;
  ...
}
```

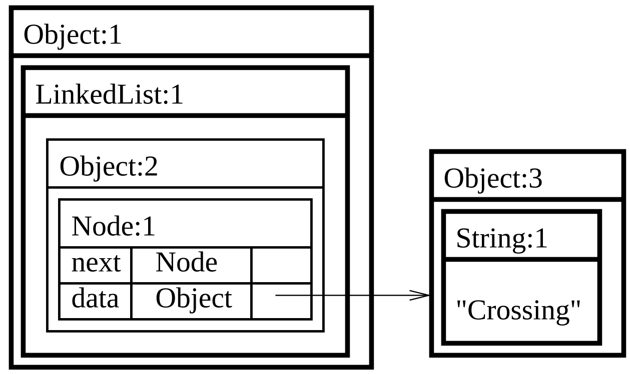


Figure 7.2: Necessary crossings in a contour diagram. An outline of the program's source code is shown on the left. The single structural link in this diagram has three necessary crossings: the boundaries of `Object:2`, `LinkedList:1` (the enclosing object), and `Object:1`. The crossing at the member table border is not counted since it can be easily avoided by placing the structural link on the cell's border, although such links are often drawn from the center by convention.

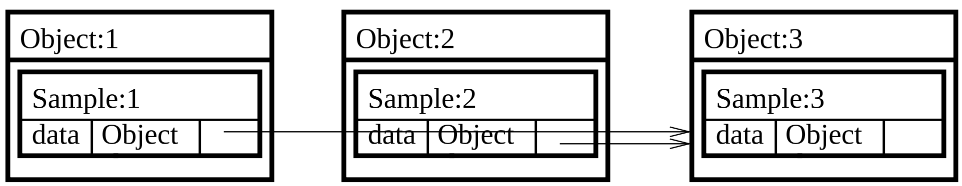


Figure 7.3: Unnecessary edge-border crossing in a contour diagram. The crossing of `Sample:1`'s `data` structural link over the `Object:2` contour group could have been avoided, for example, by placing the `Object:3` group between the other two object groups.

and meaningful, as they represent the fact that the `data` reference in `Node:1` points to an object outside of the `Node:1`, `Object:2`, `LinkedList:1`, and `Object:1` contexts. However, edge-border crossings can also be meaningless, as illustrated in Figure 7.3. The semantics of an edge-border crossing also depend on whether the edge is a structural or method return edge.

Definition 7.3. A **border-border crossing** is a crossing between two different contours' borders. Figure 7.4 illustrates a border-border crossing. These crossings can always be avoided due to the single-nesting nature of the contour model: since each contour has at most one immediately enclosing contour, the contour diagram can be drawn without any border-border crossings by ensuring that logically nested contours are properly nested in the drawing.

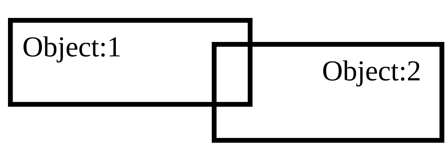


Figure 7.4: A border crossing in a contour diagram. Border crossings can always be avoided in a two-dimensional drawing of a contour diagram.

Edge-edge crossings and border-border crossings are both artifacts of a drawing; only edge-border crossings can be meaningful, although they are not necessarily so. Therefore, one must be able to distinguish between *necessary edge-border crossings* and *unnecessary edge-border crossings* in order to formalize the aesthetic preferences of our drawing methodology. Necessary crossings are those that cannot be avoided due to the structure of the contour model, as in Figure 7.2; unnecessary crossings are those that are artifacts of the drawing, as in Figure 7.3. This concept is further explained and formalized below.

Let v be a variable defined within the member table of a contour c_k, where c_k is contained in a contour group $C = c_1, c_2, \ldots, c_k$ such that c_{i-1} contains c_i for $2 \leq i \leq k$. Furthermore, let v reference a contour d_m, where d_m is in a contour group $D = d_1, d_2, \ldots, d_m$ such that d_{j-1} contains d_j for $2 \leq j \leq m$. Furthermore, assume that C and D are immediately contained in the same context; this context may be the root of the contour model or may be a shared static or object context if C and D are inner objects. Let e be the structural link from v to d_m. In a traditional two-dimensional drawing of the contour model, e must necessarily cross the borders of contours c_k, c_{k-1}, $\ldots$, c_1, d_1, d_2, $\ldots$, d_{m-1}, until finally ending on the border of d_m. Each of these crossings is a *necessary crossing*, and all other edge-border crossings of e in the contour diagram are *unnecessary crossings*.

Although unnecessary crossings are semantically meaningless, it is not always possible to remove them. This is analogous to the general graph case illustrated in Figure 7.1: although the crossings are aesthetically bad, they cannot always be eliminated, depending on the immutable constraints of the drawing.

Nested Structures

One of the major strengths of the contour model semantics for block-structured programs is the ease with which variable scope is represented. As discussed in Chapter 3, this is made possible by a simple scope resolution algorithm and the disciplined nesting of contours. However, this nesting is a major reason that contour models cannot be treated as simple graphs for the purpose of drawing contour diagrams.

Contour groups can be represented as nesting trees, which presented in Section 2.4. This enables the use of HV-inclusion drawings to draw contour groups. However, constructing an HV-inclusion drawing with minimum area is NP-hard [63]. Additionally, an HV-inclusion drawing does not necessarily capture the semantics of any nested structures. For example, a binary search tree can

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

Listing 7.1: Binary search tree with inner object data structure.

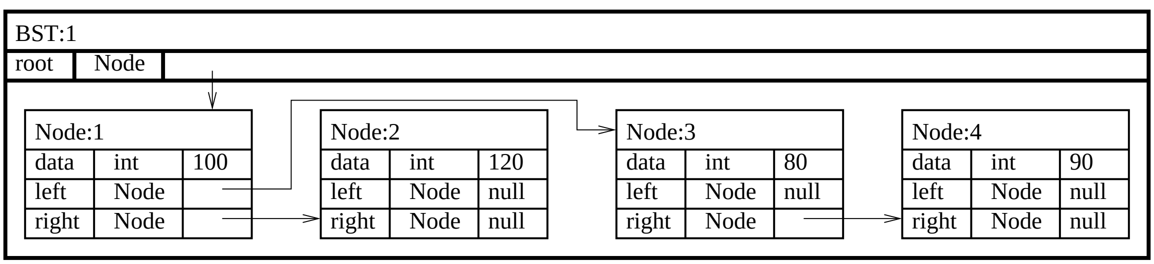


Figure 7.5: HV-inclusion drawing of an inner object binary search tree.

be designed so that the tree nodes are fully encapsulated inner objects of a top-level tree class. Listing 7.1 provides an example of this type of structure, and Figures 7.5 and 7.6 present two drawings of the same state of this program's execution. Figure 7.5 uses an HV-inclusion drawing, which takes minimal area but does not highlight the object structure. Figure 7.6 uses a more traditional rendering of a binary tree and therefore consumes much more area, but it better captures the "meaning" of the program state in a fundamental sense.

Nesting a top-level drawing algorithm within the limited canvas of a contour can potentially waste a great deal of space, as illustrated in Figure 7.6. However, when there is such a substructure, it is in keeping with our desiderata and aesthetic preferences to draw the structure as shown. The HV-inclusion drawings save space, as shown in Figure 7.5, but do not reflect any inherent structure.

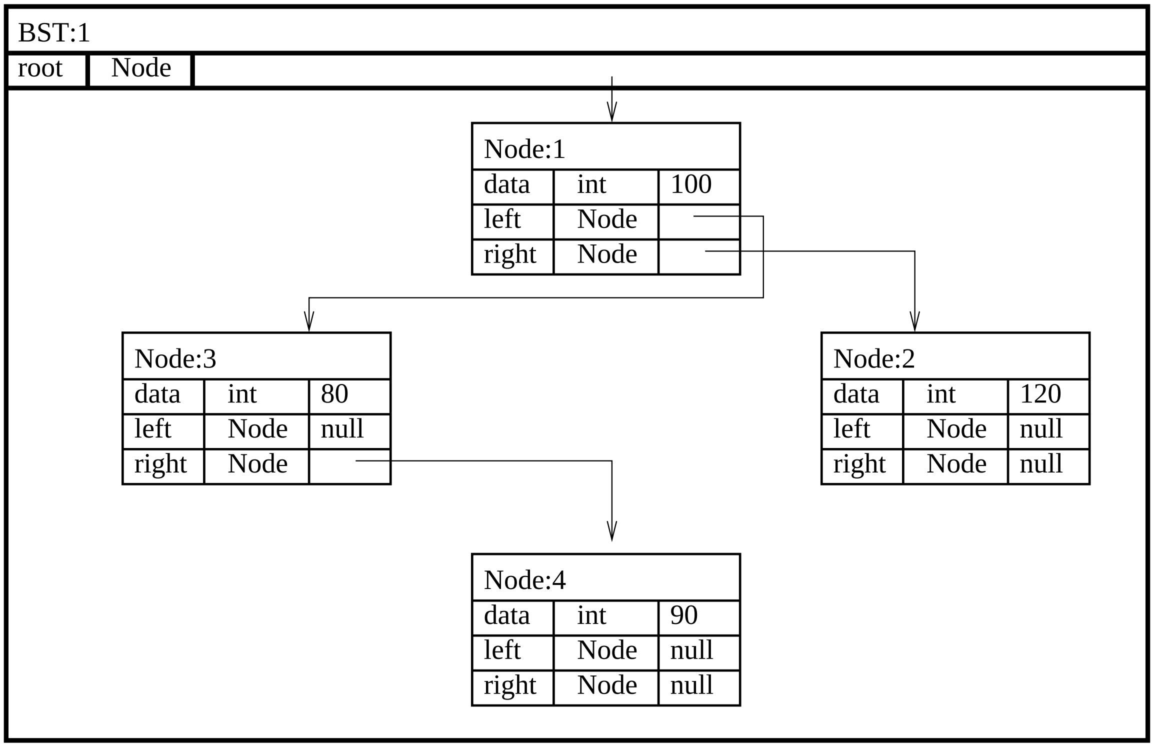


Figure 7.6: Traditional drawing of an inner object binary search tree.

We observe that both inner object contours and method contours can be nested within a contour, and the former may have a "structure" while the latter may not. Hence, our approach in the JIVE methodology is to draw method activations using an HV-inclusion drawing, and to draw nested object structures by treating the inner object context as its own drawing area. In this approach, links that cross drawing area boundaries are ignored for the purpose of drawing the nested structure.

Heterogeneous Edges

There are two types of edges that are drawn explicitly in a contour diagram: structural links and return links. *Static links*, mentioned briefly in Chapter 3, are implicit in the contour model and not drawn in a contour diagram. Visually, return links are color-coded according to their thread, and structural links are shown in a separate color. This implies that in a system with t threads, there are $t + 1$ types of edges to consider when drawing the contour model. In any specific contour diagram (that is, in any particular state of the contour model), there are $s + 1$ types of edges to consider, where s is the number of concurrently executed threads; note that $s \leq t$ since t is the maximum number of concurrent threads.

Given that there are multiple heterogeneous types of edges, we can further categorize edge-edge crossings as follows:

- **Heterogeneous crossings** occur when two different types of edges cross.
- **Homogeneous crossings** occur when two edges cross that are the same type.

Aesthetically, homogeneous crossings are worse than heterogeneous. This is because these edges are rendered in the same color, and hence there is increased chance of visual confusion between edges; this is in contrast to heterogeneous edge crossings, where edges are rendered in different colors, and hence there is a reduced chance of visual confusion. This visual confusion, in which a reader cannot easily follow or understand edges, is the primary reason for the preference for reduced edge crossings in general graph drawing, and this is drawn from the human perception studies such as those of Batini et al. [11]. We can conclude then that a homogeneous crossing is, in a sense, a traditional crossing that complicates a diagram.

Drawing Conventions

We observe that there is a tendency for object-oriented systems to produce hierarchical structures. By this observation, we do not mean to imply that object-orientation necessarily yields graph-theoretic trees from the object references on the heap; rather, we observe that networks of objects tend towards decomposition in size and complexity towards leaf nodes. This observation is supported by the recent work of Potanin et al. [73], who have shown that the distribution of objects with respect to object references follows a power law: there are few objects with high degree and many objects with low degree. Furthermore, UML class diagrams are generally drawn in a tree-like fashion, with superclasses shown above subclasses; our desire to help close the loop between design and runtime understanding pushes us towards a mirror of this structure.

An example of this phenomenon occurs in the construction of graphical user interfaces using the Swing library. Like many other common GUI libraries, Swing interfaces are composed by nesting components within containers, following the Composite design pattern [32]. Figure 7.7 provides a class diagram for part of the Swing API, and Figure 7.8 provides a simple object diagram illustrating the general hierarchical shape of the object structure. This figure does not show all of the fields and references involved in the object structure; it only provides a skeletal overview. Figure 7.9 builds upon this by including more of the details that might be present in an actual GUI, such as layout objects (*e.g.* `Box`) and control implementations (*e.g.* `Action`). The object diagrams of Figures 7.8 and 7.9 have been drawn using a layered drawing strategy.

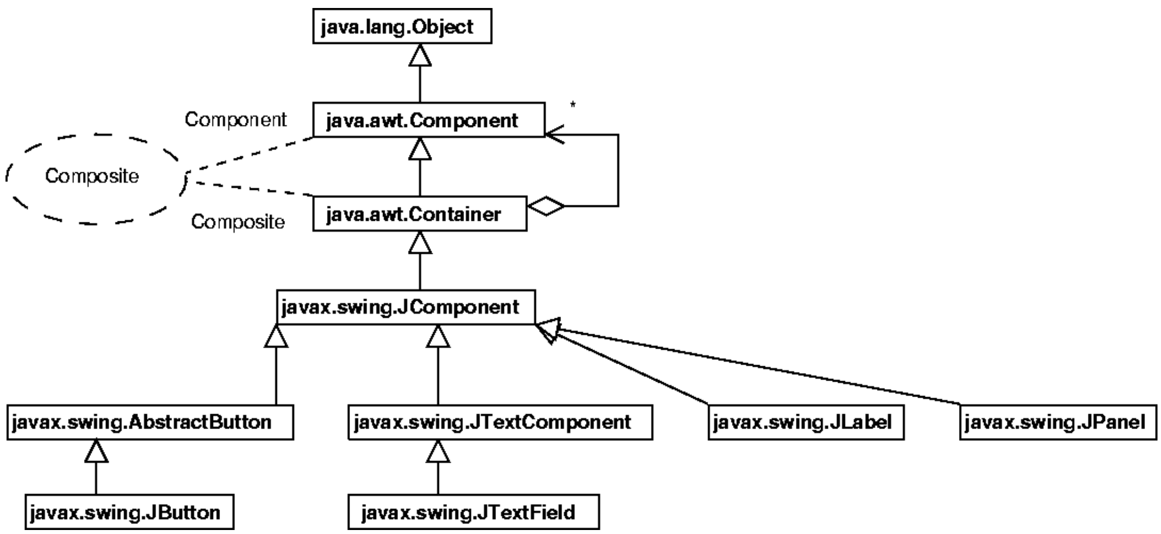


Figure 7.7: Class hierarchy for part of Java's Swing API

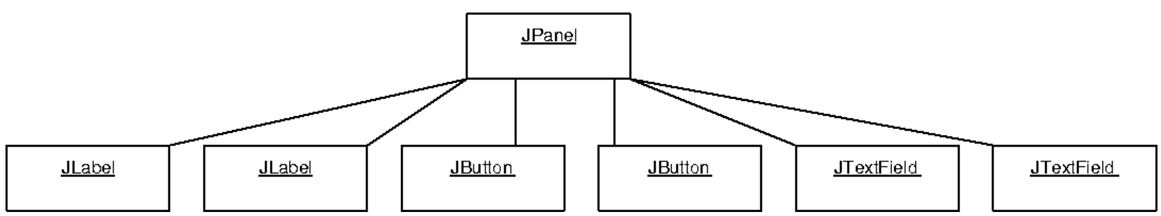


Figure 7.8: Simplified object diagram for a simple Swing GUI

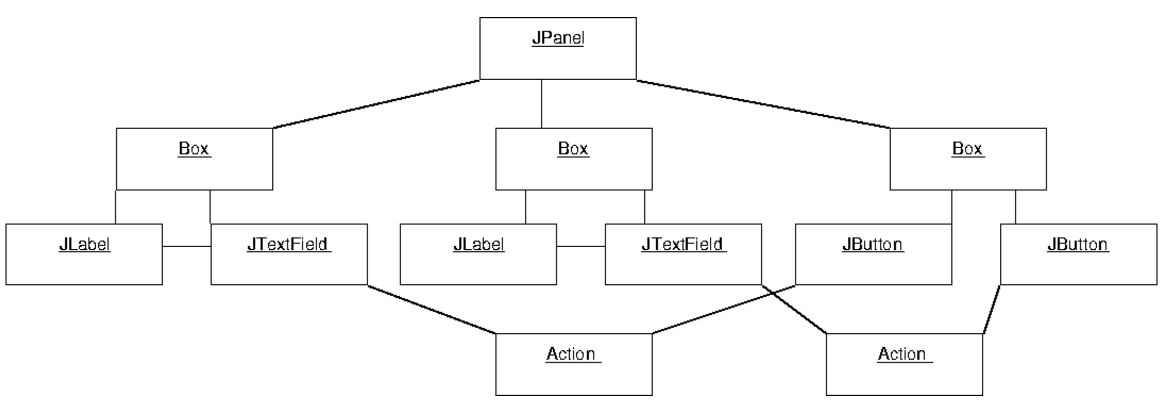


Figure 7.9: A more accurate object diagram for a simple Swing GUI. This diagram shows some objects whose classes are not shown in Figure 7.7.

Summary

Our aesthetic preferences for our drawing, based upon the observations above and the particular properties of the contour model, include the following:

1. Layeredness
2. Zero border-border crossings
3. Minimized unnecessary edge-border crossings
4. Minimized homogeneous edge-edge crossings
5. Upwardness
6. Centeredness
7. Minimization of area
8. Minimization of bends
9. Minimization of total edge length
10. Representation of logical object structure

Given these aesthetic preferences, a hierarchical layered drawing approach is reasonable since such a technique preserves layeredness and prevents border-border crossings. To draw nested structures, the same technique will be applied recursively within contours that contain inner objects.

There are two general approaches that can be taken at this point: either custom algorithms can be crafted to perform hierarchical drawing on a contour model directly, or the contour model can be embedded as a graph, allowing standard techniques to be used. The latter is preferable since it allows more direct application of the vast amount of existing work on graph algorithms and hierarchical drawing algorithms in particular.

7.2 Embedding a Contour Model as a Graph

The motivation for embedding a contour model as a graph is to facilitate the application of known algorithms for drawing a contour diagram. The technique can be summarized by the following three steps:

1. Convert a contour model M into a graph G.

2. Generate a drawing Γ of G.

3. Transform Γ into a contour diagram by reversing the transformation of step 1.

Hence, the transformation from a contour model into a graph must be a reversible transformation. We do not require that the transformation be lossless as long as the transformation is reversible; any information that is lost in the transformation can be stored outside of the graph and re-introduced during the reverse transformation. The most important property of the transformation is that it yield a graph whose drawing can be easily converted into a contour diagram that satisfies the maximal set of aesthetics.

Capturing the object structure

It is desirable that a contour diagram capture the inherent object structure of a state of program execution. However, it is not immediately clear how this "inherent object structure" can be represented. We define two techniques for extracting this structure:

Persistent structural link technique: In this approach, only those structural links from object and static contours are considered. Conceptually, structural links from object contours form the skeletal object structure of a system. Structural links from static contours are often used to represent constants or similarly persistent structures. On the other hand, method contours are inherently transient, as are structural links emerging from them. Method structural links, which can come from parameters or local variables, are omitted from consideration in this approach due to Property 7.1: since methods are inherently transient, any structural links emerging from them must also be.

Total structural link technique: Under this approach, all structural links are considered part of the inherent object structure. This includes links emitted from object, static, and method contours. As mentioned above, the structural links emerging from object and static tend to form more persistent structures. This approach's additional use of method structural links is is important for capturing transient substructures formed by methods. One advantage of this approach is in its capacity for predicting object structures when mutator methods are used to set fields' values, as demonstrated in Figure 7.10. When reference values are used, the parameter of such a "setter" method forms a temporary link to a target object that is imminently assigned to a field.

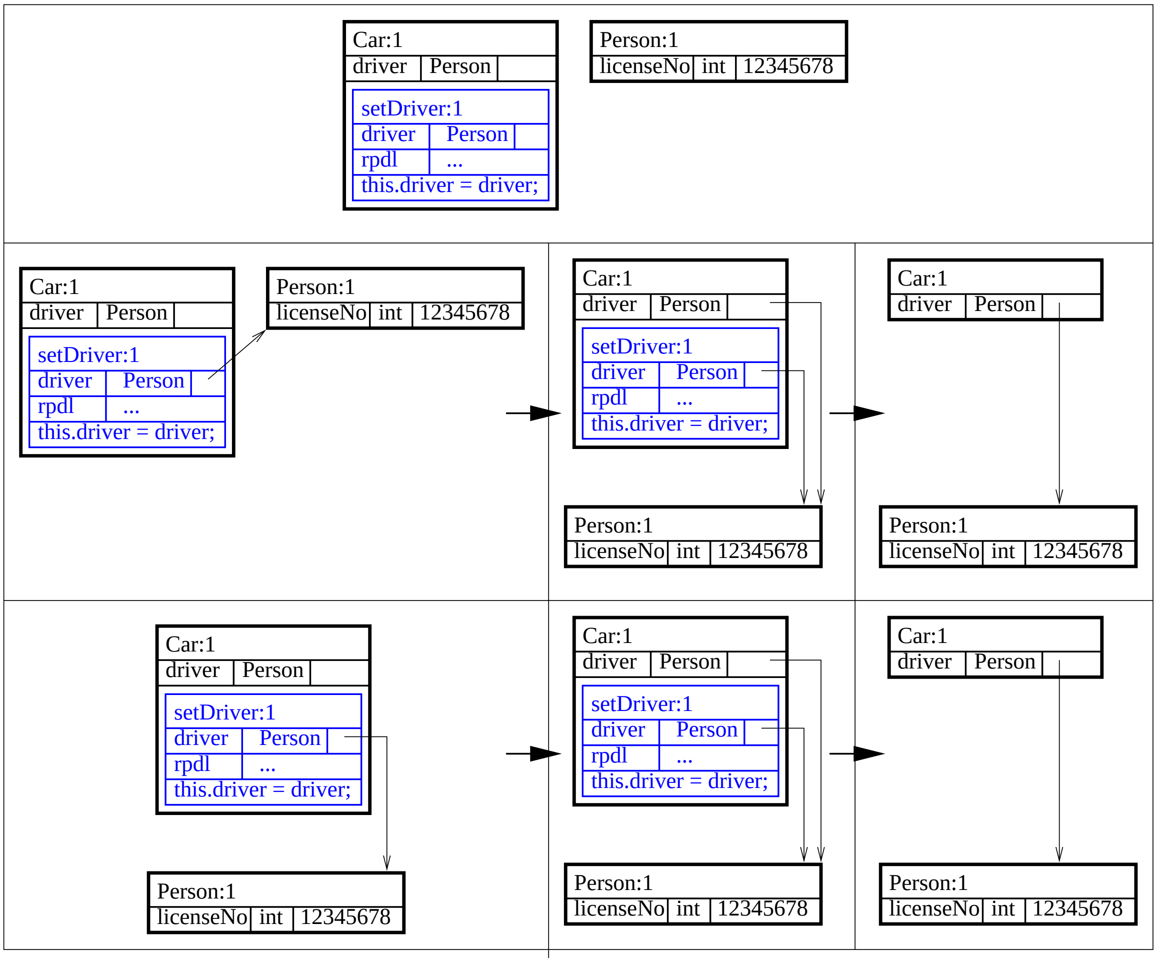


Figure 7.10: Illustration of the persistent structural link approach versus the total structural link approach to determining object skeletons. In this example, a mutator method `setDriver` is being called on `Car:1` to set its `driver` field to `Person:1`. The initial state, where there are no links between these two objects, is shown at the top of the figure. The middle row illustrates the persistent structural link approach: the link from `setDriver:1` is not considered in the hierarchical layout (*i.e.* layer assignment) of the leftmost figure. The bottom row illustrates the total structural link approach, where the link from the method is used in layer assignment, and it is clear that the use of the method link helps maintain a consistent model since the method structural link antecedes the object structural link.

Clearly, neither technique will be always be correct in capturing the inherent object structure. The preferable technique depends upon the nature of the program being visualized, and even within a single program, different techniques may yield better results at different times. Both techniques ignore method return links for all threads. The motivation for this omission is that methods are transient; an object-oriented approach tends to focus on the structure of objects, while many relatively small methods are activated in the various object contexts. In the current JIVE prototype, we have used the *persistent structural link technique*. This decision is based primarily on the desire for less complex graphs, although experimentation with other techniques has been planned as future work (see Chapter 10).

A problem similar to the one above arises in the consideration of method return links. Methods are transient, and so like their structural links, their return links also are removed from the model upon method termination. Method return links do not of themselves add to the object structure of network of objects; they are behavioral or control links, not structural links. However, there are cases where it would be desirable to utilize method return links for constructing a drawing. The most prominent example arises when there are methods on multiple threads of execution within a single object context, where the methods have no structural links incident upon them. Rather than treat all methods as equivalent, it would be useful to gather methods of the same thread, and this would be done automatically if return links were considered in the structure, as shown in Figure 7.11. Notice, however, that it is desirable to reverse the direction of return links so that upwardness is maintained in the same direction as with structural links.

The inherent problem with treating method return links as part of the underlying graph structure is that there is a high probability of objects drastically changing position as the program executes. Object-orientation encourages the use of many small methods such as accessors and mutators. Figure 7.12 illustrates the problem where an object is temporarily assumed to be part of a structure due to the adopted method return link processing technique. Due to the frequency with which this situation arises in practice, based on experimentation with the JIVE tool, we do not treat method return links as part of the graph structure in the conversion from the contour model to a graph.

A final consideration regarding capturing the object structure within a graph is that of object coupling. The coupling of objects can be measured as the number of structural links between them, not including those emitted from method contours. It is desirable that highly coupled objects be drawn in proximity to each other, and this preference is proportional to the degree of interobject coupling. Coupling data can be retained in the graph by using a multigraph representation, where the number of structural links between contour groups corresponds directly to the number of edges

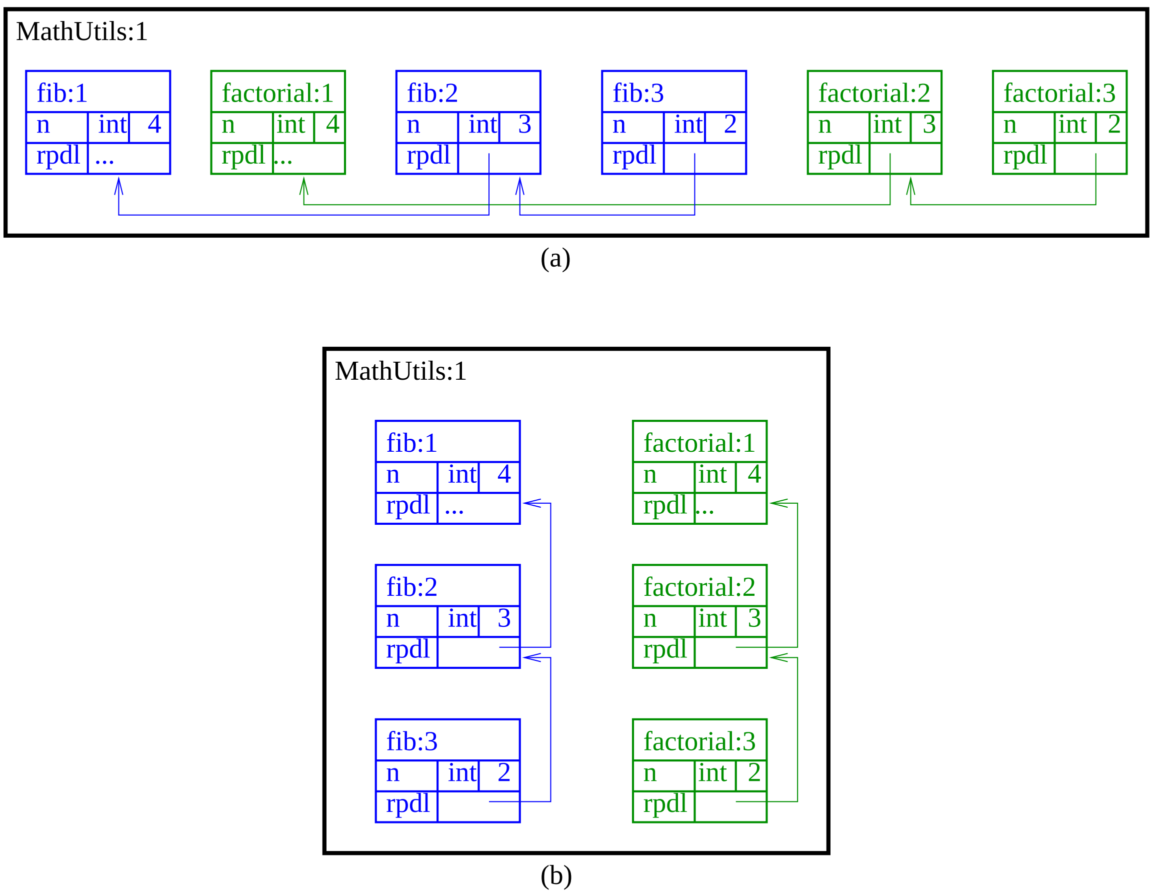


Figure 7.11: An illustration of two approaches to processing method return links. Both drawings represent the same data and use a layered drawing approach. Part (a) does not consider method return links as part of the underlying graph representation, whereas part (b) does.

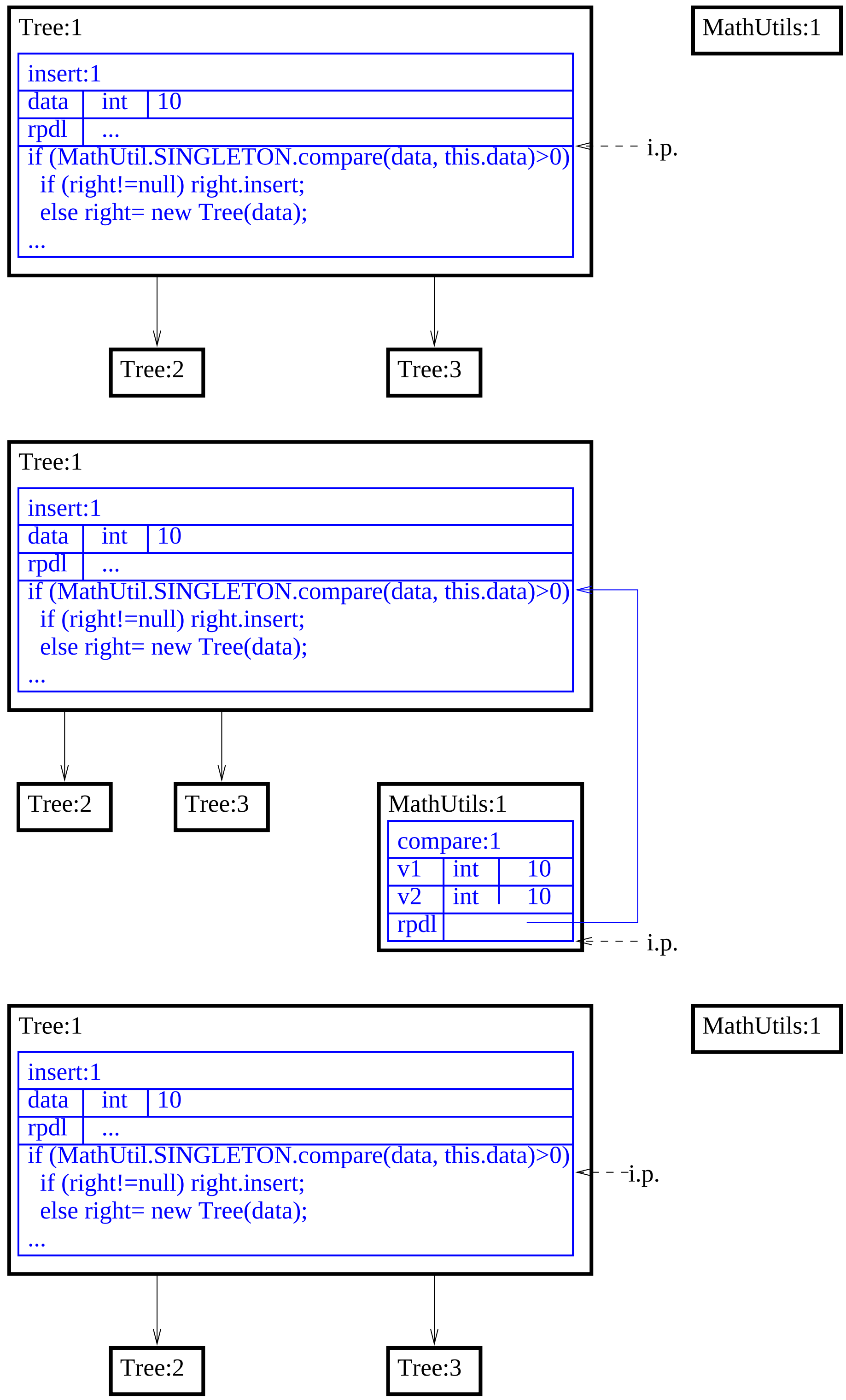


Figure 7.12: An illustration of the complication of processing return links as structural links. The `MathUtils:1` object is not conceptually a part of the tree structure. However, if return links are processed as structural links, then it temporarily "joins" the tree structure, but only for the duration of its `compare` method execution; then it returns to its original, external placement.

between the corresponding nodes. The implications of using multigraphs is discussed following a presentation of the embedding algorithm itself.

Embedding Algorithm and Analysis

Using the persistent structural link technique for extracting the inherent object structure from a contour model, combined with the intentional elision of method return links, implies that method contours themselves are essentially removed from consideration. The technique for embedding a contour model as a graph involves "collapsing" contour groups into single nodes. Recall that a contour group can be an object group or the entirety of static space. As a contour group is collapsed into a single vertex, the structural links incident upon all contours in the group become edges incident on the vertex. That is, given a contour group $G = \{c_1, c_2, \ldots, c_k\}$, where c_i is a contour in the group G for $1 \leq i \leq k$, the group G is converted into a vertex v. Then, for each structural link incident upon any $c_i \in G$, an edge is created whose endpoint corresponding to c_i is v. The resulting graph is the *contour graph* for the given contour model. Algorithm 7.1 presents a formal technique for performing the conversion in two passes through the contour model: in the first pass, contour groups are essentially collapsed into vertices; in the second pass, structural links are converted into edges.

Input: Contour model M
Output: (Directed multigraph, Mapping of contours to vertices)

```
P ← empty map;
V ← ∅;
E ← ∅;
foreach root contour c in M do
    let v be a new vertex;
    add v to V;
    add mapping (c, v) to P;
    foreach  contour d nested at any depth in c) do
        add mapping (d, v) to P;
    end
end
foreach structural link (c, d) in M do
    if c is not a method contour then
        u ← mapping of c in P;
        v ← mapping of d in P;
        add (u, v) to E;
    end
end
return (G = (V, E), P)
```

Algorithm 7.1: EmbedContourModel: Embed a contour model as a graph

This approach takes $O(|C| + |L|)$ time, where C is the set of contours in the contour model and L is the set of structural links in the contour model. Notice that the algorithm returns the graph as well as the mapping of contours to vertices; this mapping is required in order to process the contours directly after the drawing Γ has been produced from the graph G.

Most hierarchical graph drawing algorithms operate over acyclic graphs. It is not unusual to have a cycle removal step prior to producing a drawing, and our technique is no exception. However, the use of multigraphs to encode contour models requires special consideration. Using multigraphs enables the direct measurement of coupling when removing cycles from a graph. Figure 7.13 illustrates how undesirable layerings can be obtained if coupling data is ignored. In this example, there are two strongly coupled object contours, `A:1` and `B:1`. The contour model is converted into a graph by a simple, non-multigraph embedding, and so during cycle removal, the edge between `A:1` and `B:1` is arbitrarily chosen to reverse. During the layering stage of the drawing algorithm (see Section 7.3), `A:1` and `B:1` are placed on distant layers, and hence their strong coupling has been lost in the drawing. Compare this situation with that of Figure 7.14, where a multigraph embedding is used, and an additional heuristic is added that, during cycle removal, edges with less weight are reversed before edges with more. As the figure shows, the coupling of `A:1` and `B:1` is maintained in the multigraph and in the design of the cycle removal algorithm, and so in the layering step, the two highly-coupled objects are assigned to adjacent layers.

7.3 Layered Drawing Algorithms

The approach used to draw enhanced object diagrams in the current implementation of JIVE uses a hierarchical, layered drawing; it is a customization of the technique described in Chapter 9 of Di Battista et al. [24]. Given a directed multigraph representation, as described in the previous section, our approach can be summarized by the following steps:

1. Assign the nodes to layers based on their topological numbering.

2. Arrange nodes within layers to minimize edge crossings.

3. Replace the drawing with a contour diagram.

The skeletal algorithm of our approach is presented as Algorithm 7.2, which contains forward references to algorithms presented later in this section. This algorithm is based on the aesthetic criteria presented in Section 7.1 and uses the contour graph embedding presented in Section 7.2.

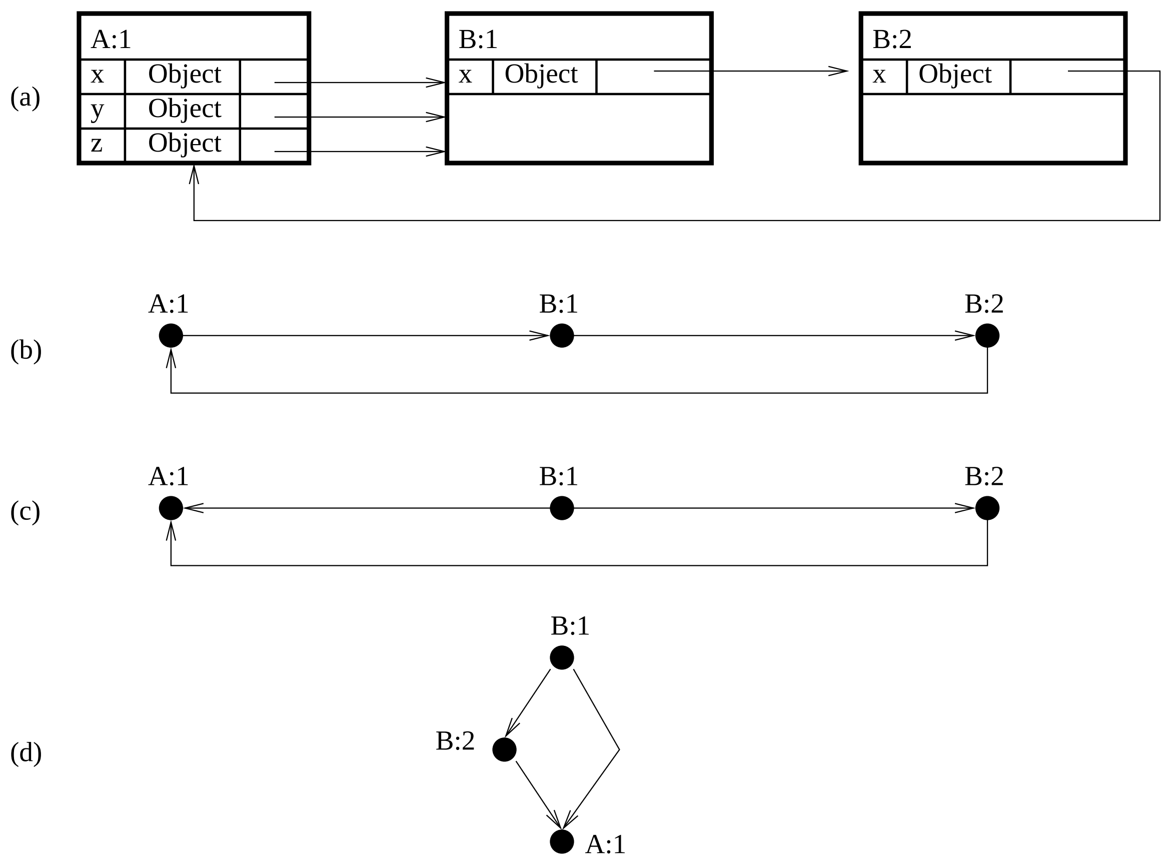


Figure 7.13: Simple graph embedding example. This example illustrates how treating all edges with the same weight can produce suboptimal drawings. Part (a) is a contour model. Part (b) shows an embedding as a directed graph. Part (c) shows the graph of (b) converted into a directed acyclic graph by reversing an arbitrary edge, in this case, the edge from `A:1` to `B:1`; notice that this edge in fact represents three structural links. Part (d) shows a drawing of the graph from (c); `B:1` and `A:1` are removed by two layers even though they are strongly coupled in (a).

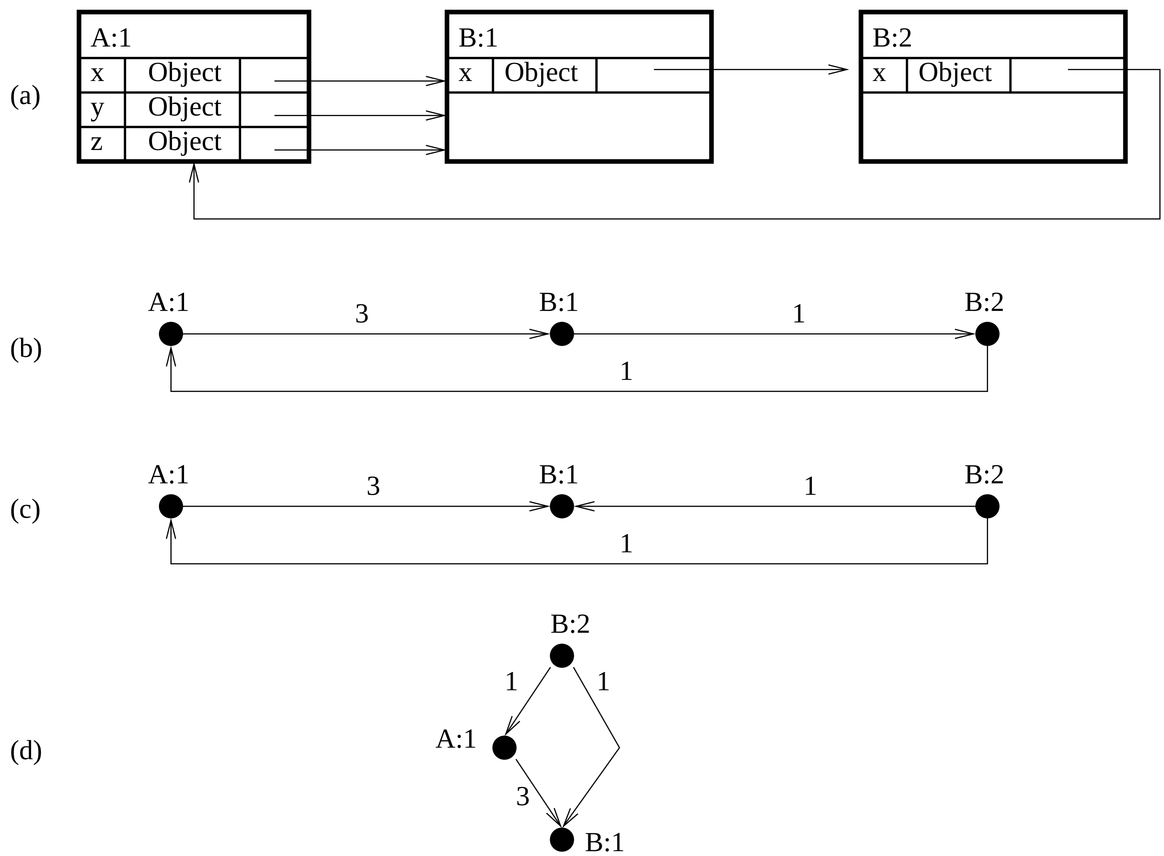


Figure 7.14: Improved multigraph embedding example. This figure improves the technique from Figure 7.13 to use a directed multigraph instead of a directed graph. The weights of each edge are indicated in parts (b), (c), and (d). In part (c), an edge with weight one has been reversed; either of the edges with weight one would be reversed before the one with weight three in this technique. The resulting drawing in part (d) has kept `A:1` and `B:1` coupled, with `A:1` dominating `B:1`, in accordance with the contour model.

Input: Contour model M;
maximum contours per horizontal layer w;
number of iterations for crossing reduction r;
vertical and horizontal spacing s_v and s_h
Output: Contour diagram

//Embed the contour model as a contour graph.
$(G, P) \leftarrow EmbedContourModel(M)$;

//Remove cycles from the contour graph.
GreedyCycleRemoval(G);

//Assign layers to the vertices of the contour graph.
$\Gamma \leftarrow$ Layer(G,w);

//Reduce crossings in the virtual drawing.
$i \leftarrow 0$;
while $i < r$ **do**
    foreach *Adjacent pair of layers* L_i *and* L_{i+1} **do**
        $x \leftarrow$ Split(Γ, current ordering of L_i);
        Assign vertex sequence x to L_{i+1};
    end
    $i \leftarrow i + 1$;
end
return *BuildContourDiagram(*Γ*, M, P,* s_v*,* s_h*)*

Algorithm 7.2: DrawObjectDiagram: Construct an object diagram from a contour model. This algorithm presents a high-level overview of our drawing technique as described in this chapter. The subroutines refer to algorithms presented later in this chapter.

Layer Assignment

Prior to layer assignment, cycles are removed by reversing a proper subset of the edge set. Finding the minimal cardinality subset to reverse is NP-complete, as it is equivalent to the *minimum feedback arc set* problem [see 34]. The cycle removal problem can be described as finding a vertex sequence $S = (v_1, v_2, \ldots, v_n)$ of a graph G. A *leftward edge* of S is an edge (v_i, v_j) where $i > j$. Given the vertex sequence S, reversing the leftward edges makes the corresponding graph acyclic. The cycle removal approach we adopt uses the *Greedy-Cycle-Removal* algorithm of Di Battista et al. [24, Chapter 9], which takes linear time with respect to the size of the input. We make minor extensions to the algorithm in order to support multigraphs. Specifically, the semantics of *outdegree* and *indegree* are enhanced for multigraph consideration. The GreedyCycleRemoval is presented here as Algorithm 7.3.

A *layering* of a graph $G = (V, E)$ is a partitioning of V into subsets $L_1, L_2, \ldots, L_h$ such that if $(u, v) \in E$, where $u \in L_i$ and $v \in L_j$, then $i > j$. The layer assignment in JIVE is performed using Coffman-Graham layering [21], an application of multiprocessor scheduling to graph drawing, presented here as Algorithm 7.4. This approach produces a layering where the number of vertices

Input: Multigraph $G = (V, E)$
Output: Vertex sequence S for G

```
S_l ← empty list;
S_r ← empty list;
while G is not empty do
    foreach v ∈ V do
        if degree(v) = 0 then
            remove v from G;
            prepend v to S_r;
        end
    end
    while G contains a sink do
        choose a sink u of G;
        remove u from G;
        prepend u it to S_r;
    end
    while G contains a source do
        choose a source v of G;
        remove v from G;
        append v to S_l;
    end
    if G is not empty then
        choose vertex u such that outdegree(u) − indegree(u) is maximum;
        remove u from G;
        append u to S_l;
    end
end
return concatenation of S_l with S_r
```

Algorithm 7.3: GreedyCycleRemoval: Use a greedy approach to remove cycles from a multigraph, a customization of the technique described in Di Battista et al. [24, Chapter 9].

Input: Directed multigraph $G = (V, E)$; positive integer w
Output: Layering of G with width at most w

Topologically number the vertices of G;
$k \leftarrow 1$;
$L_1 \leftarrow \emptyset$;
$U \leftarrow \emptyset$;
while $U \neq V$ **do**
  Choose $u \in V - U$ such that every vertex in $\{v : (u, v) \in E\}$ is in U and the label of u is minimized;
  if $|L_k| < w$ *and, for every edge* $(u, r) \in E$*,* $r \in L_1 \cup L_2 \cup \ldots \cup L_{k-1}$ **then**
    Add u to L_k;
  else
    $k \leftarrow k + 1$;
    $L_k \leftarrow \{u\}$;
  end
  Add u to U;
end

Algorithm 7.4: Layer: Assign layers to the vertices of a directed aacyclicmultigraph. The topological number can be done using a standard breadth-first traversal of the graph. This is equivalent to the algorithm presented in Di Battista et al. [24, Chapter 9], included here for completeness.

assigned to a single layer (*i.e.* in a partition) is constrained. In practice, we have used an arbitrarily large integer bound on the number of vertices per layer, which can lead to wide diagrams. The actual horizontal bound to be used is either determined by the user based on their preference for wide diagrams or guessed by the system based on the size and aspect ratio of the output device (screen, printer, *etc.*). This layering algorithm is equivalent to the one presented in Di Battista et al. [24, Chapter 9].

Minimizing Crossing

Once the vertices have been assigned to layers, crossings can be reduced by fixing the x-coordinate (horizontal placement) of each vertex on its layer. This is done using a method reminiscent of quicksort, called the *Split* method, and it is presented here as Algorithm 7.5. This approach involves selecting pivots and then moving each vertex v based on whether it produces less crossings to the left or right of the pivot [26]. Like quicksort, this approach has a worst-case complexity of $O(|V|^2)$ but, in practice, runs in $O(|V|log|V|)$ time. The algorithm operates on adjacent pairs of layers, and in practice, each pair of layers can be processed a fixed number of times to yield the best results.

Alternate techniques can be used to achieve better running times; for example, barycenter or median methods operate by placing vertices at the barycenter or median of their dominating vertices' locations, and these run in linear time [see *e.g.* 2, 24]. Integer programming techniques to minimizing

Input: two-layered directed multigraph $G = (L_1, L_2, E)$; vertex ordering x_1 for layer L_1
Output: vertex order x_2 for layer L_2

if *L_2 is not empty* **then**
  Choose a pivot vertex $p \in L_2$;
  $V_{left} \leftarrow \emptyset$;
  $V_{right} \leftarrow \emptyset$;
  foreach *vertex $u \in L_2$ such that $u \neq p$* **do**
    $c \leftarrow$ number of crossings obtained with u left of p;
    $d \leftarrow$ number of crossings obtained with u right of p;
    if $c \leq d$ **then**
      Place u in V_{left};
    else
      Place u in V_{right};
    end
  end
  return *the concatenation of the results of applying this algorithm recursively to the graphs of V_{left} and V_{right}*
end

Algorithm 7.5: Split: Reduce crossings between two layers. This algorithm is adapted from Di Battista et al. [24, Chapter 9] and is included here for completeness.

crossings are guaranteed to find the optimum solution, although there is no guarantee that they terminate in polynomial time [57]. The Split approach was chosen for the latest JIVE prototype due to the simplicity of implementation and plans for extension to dynamic graph support in the future [see *e.g.* 22]. These planned extensions will require significant modification of the current drawing algorithm, including its layering and crossing reduction steps, and so a simple and efficient algorithm was desired.

Conversion to a Contour Diagram

The drawing Γ of the contour graph G is constructed once the layers have been assigned and crossings have been reduced. The next step is to convert Γ back into a contour diagram. Recall that the contour model was converted into a contour graph, and then the drawing is constructed; converting to a contour diagram brings us back into the domain of JIVE's contour-centric visualizations. Algorithm 7.6 can be used to build a contour diagram from Γ; this algorithm uses the mapping of contours to vertices as produced by Algorithm 7.1. Converting a vertex to a contour group is a simple matter of looking up the contour group in the mapping, and this can be done in linear time with respect to the number of contours and structural links in the model. Method contours, whose structural links were not considered in the construction of Γ, are reintroduced at this step. However, contours have nontrivial and varying area, whereas the vertices of the contour graph are of trivial

Input: Layered hhierarchicaldrawing Γ of contour graph G;
contour model M;
mapping f from vertices $v \in G$ to contours in M;
minimum vertical and horizontal spacing s_v and s_h
Output: Contour diagram

$D \leftarrow$ empty contour diagram;
$y \leftarrow 0$;
foreach *Layer* L_i *of* Γ **do**
    $x \leftarrow 0$;
    $maxheight \leftarrow 0$;
    Let $v_1, v_2, \ldots, v_k$ be the vertices of L_i in left-to-right order;
    Let $c_1, c_2, \ldots, c_k$ be the contours $f(v_1), f(v_2), \ldots, f(v_k)$;
    foreach $c_i \in \{c_1, c_2, \ldots, c_k\}$ **do**
        Place contour c_i at (x, y) in D;
        $x \leftarrow x + width(c_i) + s_h$;
        $maxheight \leftarrow max(maxheight, height(c_i))$;
    end
    $y \leftarrow y + s_v + maxheight$;
end
return D

Algorithm 7.6: BuildContourDiagram: Construct a contour diagram from the drawing of a contour graph.

size. This transformation from a graph drawing to a contour diagram requires two steps: horizontal arrangement of contours and vertical placement of layers.

The horizontal arrangement of contours on a layer is determined in the layout algorithm, and the transformation to a contour diagram does not affect contours' relative positions. Let s_h be the amount of horizontal space to introduce between horizontally adjacent contours. Then a layer L that contains k contours c_1, c_2, $\ldots$, c_k is arranged by introducing s_h space between each pair of contours c_i, c_{i+1} $(1 \leq i < k)$. The amount of vertical space used by L is therefore $\sum_{c_i \in L} width(c_i) + (k-1)s_h$, where $width(c_i)$ is the width of contour c_i. The total width of the contour diagram is therefore the maximum width of any layer in the drawing. Layers are arranged vertically using a similar technique. Let s_v be the amount of vertical space to introduce between each layer. For each layer L, the height of the layer is equal to the maximum height of any contour in the layer. Let n be the number of layers in the drawing. The total height of the drawing is therefore $\sum_{layer L} height(L) + (n-1)s_h$.

The drawing strategy used in the latest JIVE prototype does not perform any advanced edge routing. Both structural and return edges are drawn as simple straight lines. The only special processing that is performed involves the placement of edge endpoints: since contours are not generally simple points (modulo minimized diagrams), the edges are drawn so that they are the shortest distance between *anchors*. For structural links, the anchor is the corresponding variable's value entry in the member table, or, if the member table is not visible, the border of the contour. For return

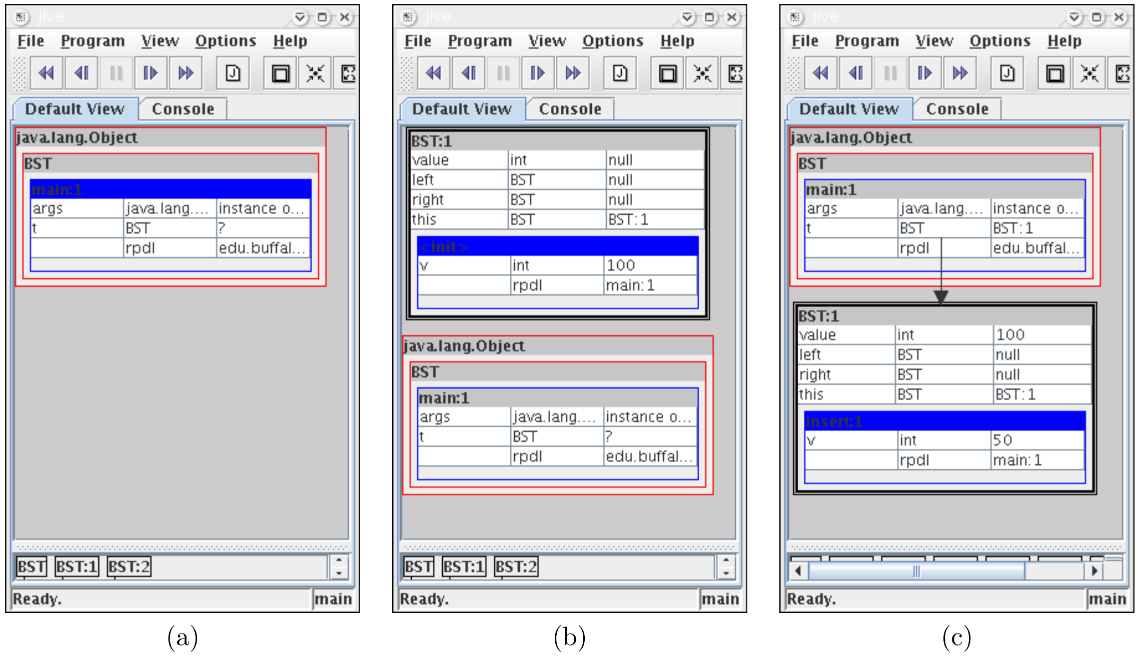


Figure 7.15: Three screenshots of JIVE illustrating the drawbacks of the current automatic drawing implementation (as of JIVE-0.26). The three shots correspond to stepping through program execution with time increasing from left to right. The relative positions of static space and the `BST` object exchange from state (b) to state (c) when the structural link is added

links, the anchor is the `rpdl` field of the member table, or, if the member table is not visible, the border of the contour. These anchor points can be computed in constant time for any connected pair of contours.

Summary and Observations

The techniques with which we have experimented in JIVE were originally developed for static (*i.e.* non-dynamic) graphs. Drawings of individual object diagrams are good and tend to meet our aesthetic constraints; however, there is no necessary correlation between subsequent diagrams, and hence there is a danger of a user losing his or her mental map of the drawing [27]. Notice, however, that although there is no necessary mathematical correlation between states, the dynamic behavior of the system has been considered in the very construction of the embedding and drawing algorithms and hence has not been totally ignored.

Figure 7.3 exemplifies the problem with three screenshots of JIVE. The figure contains three screenshots of JIVE illustrating a progression through program states. In part (a), there is only one root contour: that of static space. When the `main` method creates the first object, both root objects have indegree and outdegree of zero; since this implementation does not distinguish between object and static contours, the static contour arbitrarily ended up below the object contour. In the next discrete state, a structural link is created from static space to the object, and so the diagram is reorganized to satisfy upwardness. This is akin to the return link processing problem described along with Figure 7.12.

The good drawings of individual states generated automatically by JIVE indicate that our approach is successful since they meet many of our aesthetic preferences reasonably well. Extension to dynamic graphs requires specification of additional aesthetic constraints and enhancing the presented techniques, and so this is left as an area of future work.

Chapter 8

Drawing Sequence Diagrams

A sequence diagram is a visual model of program execution history that clarifies the sequence of interactions among objects. These diagrams are commonly used during the design process to specify desired behavior, and it has been shown that UML sequence diagrams are an effective method for expressing dynamic system behavior [44]. Like many of the tools of the Unified Modeling Language, sequence diagrams can be used at a varying levels of abstraction in order to show the the messages sent among subsystems. However, the same graphical syntax can be used to specify method-level interactions, and this is the domain in which we are using the diagrams: the representation of actual program executions rather than the specification or documentation of desired or idealized execution.

There are many tools that facilitate drawing sequence diagrams, including IBM Rational Rose, Microsoft Visio, Gentleware Poseidon, and the open-source tool Dia. These are all graphical tools through which a user manually arranges graphical elements to form sequence diagrams. In the JIVE methodology, sequence diagrams are automatically generated from a program's execution, and so these manual editing techniques are not applicable. Instead of manual layout controls in, JIVE requires efficient and effective automatic drawing algorithms. Although there has been interesting work in automatic drawing of other UML diagrams such as class diagrams [43], we know of no existing work in automatic or incremental drawing of sequence diagrams from execution trace data. In this chapter, we describe the syntax and semantics for the sequence diagrams used in JIVE and the methodology by which we support their automatic drawing. The remainder of this chapter is organized as follows:

- Section 8.1 describes sequence diagrams and their semantics in the JIVE methodology.

- Section 8.2 presents a formal analysis of the aesthetic preferences for automatically drawing sequence diagrams. This section describes how the essential drawing problem can be divided into two independent problems: arranging methods on a lifeline and arranging lifelines in a

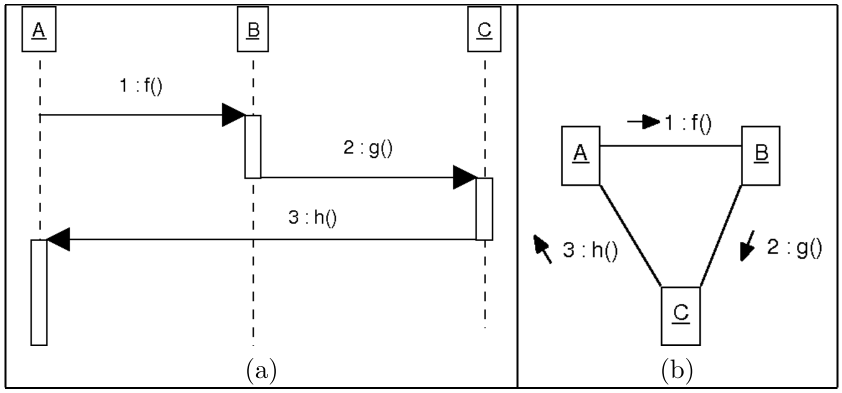


Figure 8.1: A sequence diagram and a collaboration diagram that depict the same interaction

diagram.

- Section 8.3 describes the problem of arranging methods along a lifeline along with our solution strategy.
- Section 8.4 describes the problem of horizontally arranging lifelines within a sequence diagram along with our solution strategy.

8.1 Sequence Diagram Syntax and Semantics

In the Unified Modeling Language, sequence diagrams are one form of *interaction diagrams*, the other being *collaboration diagrams*. Interaction diagrams depict *interactions*, where an interaction consists of a set of objects, their relationships, and any messages passed among them. Sequence diagrams and collaboration diagrams are representations of the same metamodel, though their focus is different. Sequence diagrams focus on the sequence of messages over time, whereas collaboration diagrams focus on the connections among objects. Figure 8.1 illustrates sequence and collaboration diagrams for the same interaction. The major difference between the two is that a sequence diagram depicts execution history with time increasing downward, and it does not show the links between objects, whereas a collaboration diagram does depict interobject links but it does not have a unidirectional flow of time.

The UML defines a rich syntax for sequence diagrams, including support for synchronous and asynchronous messages as well as stereotyped creation and destruction messages. However, the application of UML sequence diagrams is traditionally for program design, specification, and documentation, not for representation of actual program execution. By contrast, JIVE's methodology requires the visualization of actual program execution. We use the following elements of the UML

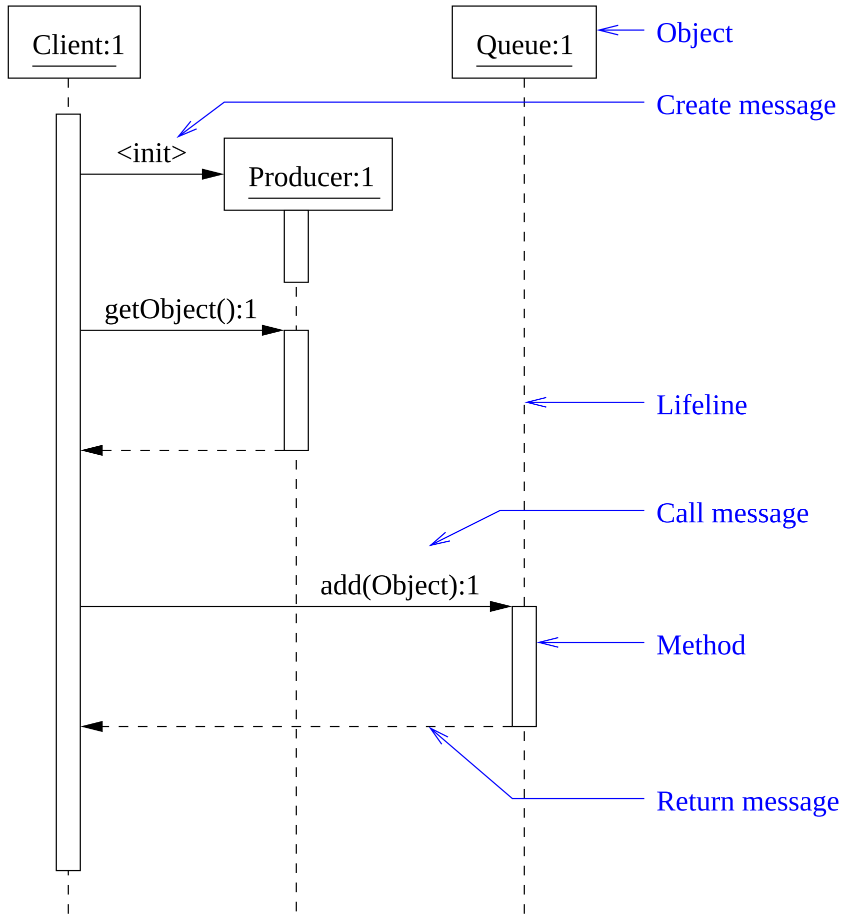


Figure 8.2: Sample sequence diagram.

sequence diagrams in the JIVE methodology, as illustrated in Figure 8.2.

Objects: As in UML sequence diagrams, *objects* are listed horizontally across the top of a sequence diagram. Objects are rendered as labeled rectangles. In this context, an "object" can be a proper Java object or a static context. These static contexts correspond to static contours in the contour model and class objects in the Java Virtual Machine. Within the contour model, object and static contexts are the contexts in which method activations can be nested. The same identifiers are used for objects in the sequence diagram as are used in the contour model, and for object groups, the innermost label is used. For example, if a contour `String:2` is nested within a contour `Object:2` in the contour model, then this object group is represented as "`String:2`" in the sequence diagram, even if method activations take place in the `Object:2` context in the contour model. Figure 8.3 demonstrates the correlation between a contour diagram and the contexts of a sequence diagram.

The vertical position of an object corresponds to the time of its creation. Those objects that

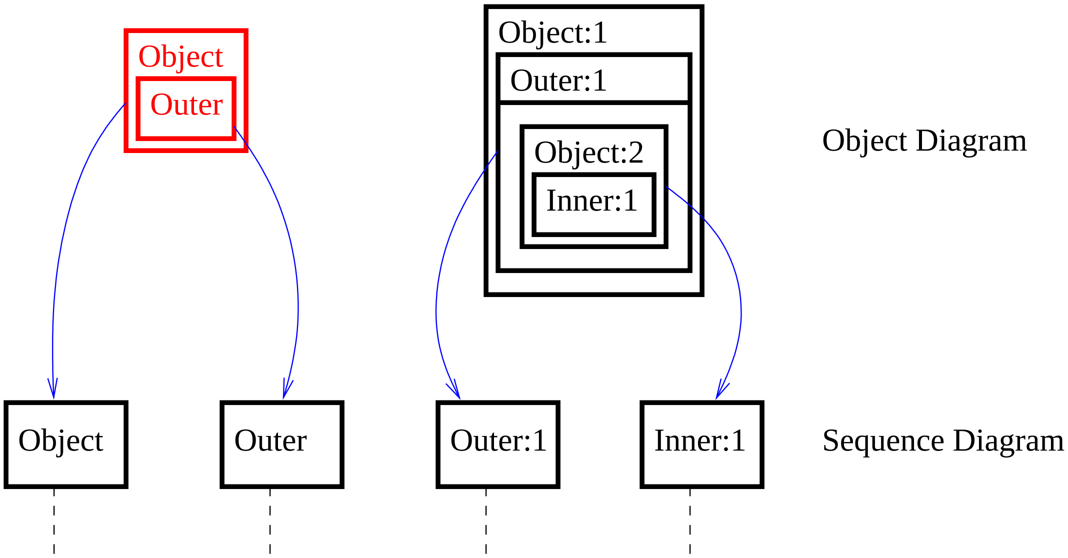


Figure 8.3: Converting the contexts from an enhanced object diagram into a sequence diagram. `Inner` is an inner class of `Outer`, and both directly extend `Object`. Static contours are shown in red and object contours in black. The blue arrows depict the translation from the object diagram to the "objects" of the sequence diagram. `Inner:1` is an instance inner class and hence has no static counterpart (see Section 3.2).

have no incoming creation message are assumed to be created by the system or by a context that is not monitored, and so these are vertically aligned at the very top of the sequence diagram. Objects that are created after system initialization most often have initializers that run immediately after creation, and although the caller of such an initializer is the system itself, it is semantically appropriate to consider it to be the context that created the object. This is because initializers cannot be invoked in any other way in Java except by subclass constructors or polymorphic constructors in the same class, and hence there is no loss of generality in our depiction of the caller as the object creator. However, some JIVE prototypes used to generate screenshots in this document have not implemented this visual semantics, and so objects are occasionally shown at the top of the sequence diagram regardless of creation time, as shown in Figure 8.4.

Inner objects are depicted as separate, independent objects in the sequence diagram. They cannot be nested within their enclosing context since the time of creation for each inner object is potentially different, in addition to the questionable visual semantics of collapsing inner objects into their enclosing objects. Drawing inner objects within their enclosing contexts would require either non-rectangular representation of objects or exceedingly large enclosing object rectangles that cross other lifelines, messages, and methods. Neither of these options is

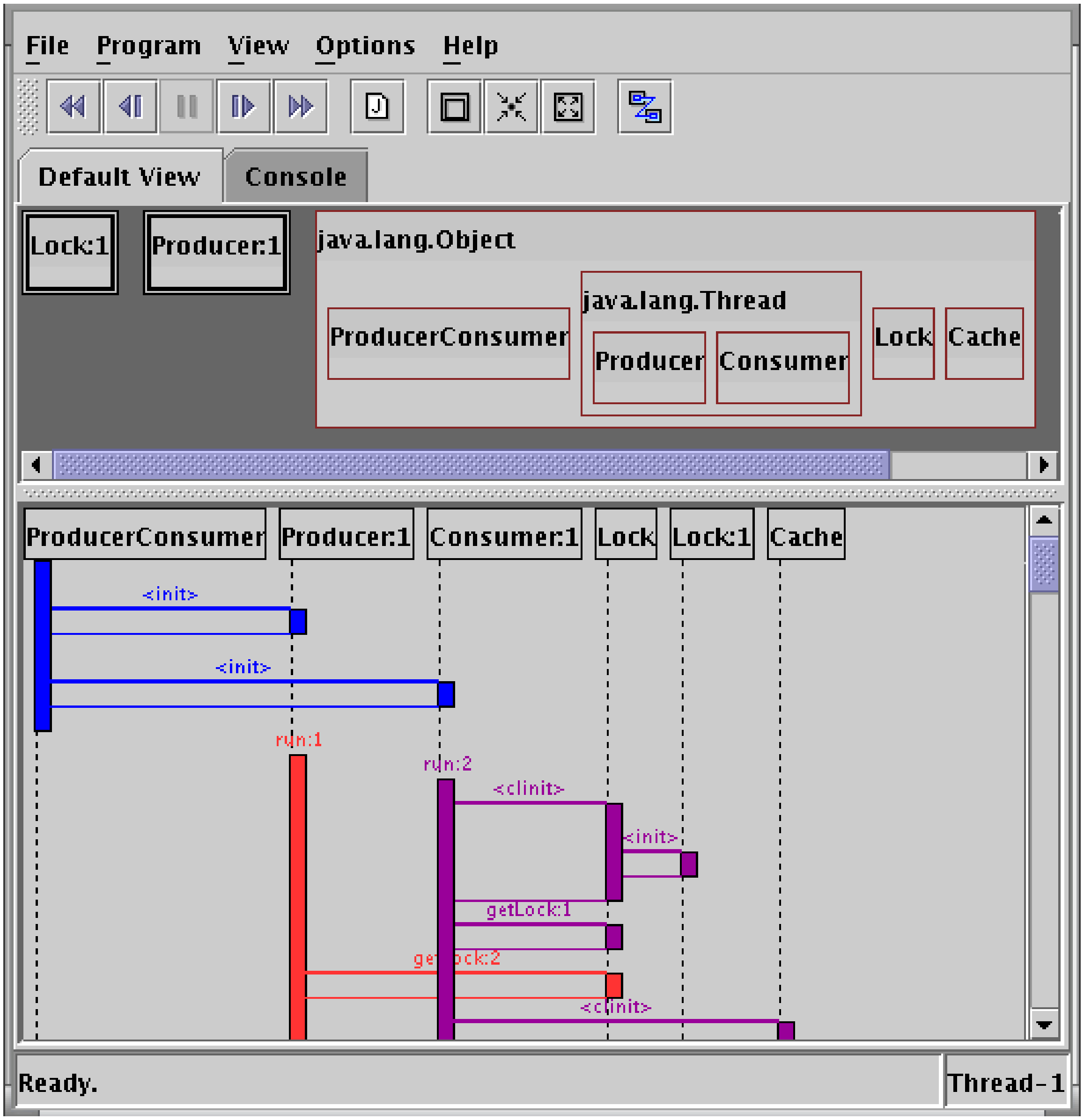


Figure 8.4: A JIVE screenshot of a multithreaded producer/consumer program. The main thread (blue) starts two other threads: `Producer` (orange) and `Consumer` (purple). The sequence diagram shows how the main thread terminates before the other threads are actually started. The producer starts before the consumer, but the consumer is the first to create and acquire a `Lock` object. The top portion of the screenshot is JIVE's representation of the object structure and static space at the end of program execution.

aesthetically pleasing, and hence inner objects are treated as separate objects. The nesting of inner objects within enclosing contexts is left to the state visualizations of contour diagrams. The JIVE methodology supports easy navigation between contour and sequence diagrams, which provides the user with the important semantic connection between the corresponding contexts.

Lifelines: A *lifeline* is emitted downward from each object, rendered as a dashed line. Time increases in the downward direction, and so a lifeline depicts the active lifetime of a sequence object, starting at object creation and extending theoretically to object destruction. In the UML, lifelines can be terminated by destruction markers, shown as an “X” through a lifeline. There are no destruction markers in JIVE’s sequence diagrams since Java objects cannot be explicitly destroyed and the JIVE architecture does not support destruction monitoring.

Methods: While rectangles along a lifeline in a UML sequence diagram may represent focus of control (an abstraction of one or more method activations) or duration of actual method execution, those of JIVE sequence diagrams exclusively represent the duration of a method’s execution. The top of the rectangle is at the time of the method’s activation, and the bottom is at the method’s exit either by a return, an exception’s being thrown, or a `void` method’s normal termination. A method rectangle therefore stretches uninterrupted from method call to method return; this is in contrast to some UML sequence diagram renderings, where the focus of control is removed from an object when it yields control to a method it has called. The width of a method rectangle is configurable based on the size of the visualization and the display metrics.

Messages: Messages depict method activations and method returns by highlighting the caller and the receiver object. The name “message” is reminiscent of Smalltalk, which *sends messages* to objects rather than *calling methods* on them. Messages are rendered as horizontal arrows, with calls being solid and returns being dashed. However, in some JIVE prototypes, thick and thin lines are used instead, as shown in Figure 8.4. Calls are annotated with the name and activation count of the method being called, where the activation count comes from the contour model. Returns are not annotated since their corresponding method is easily distinguished in the diagram; again, this is a variation from UML, in which foci of control can be intermittent, unlike methods, which are continuous in Java with respect to their thread. Creation messages are annotated as `<init>` messages, the name taken from the reserved method identifier used within the Java Virtual Machine to reference initializers. This annotation is semantically equivalent to the ≪create≫ stereotype in UML sequence diagrams. Unlike UML sequence diagrams, JIVE sequence diagrams do not differentiate between synchronous and asynchronous messages. This differentiation is necessary in the specification of system behavior, but at the runtime, synchronization mechanisms are already in place, and the thread scheduler determines the order of execution.

We augment the standard sequence diagram notation by adding color indexing for threads. Each thread is assigned a unique color, and all methods and messages on that thread are drawn in its color. This technique has proven to be useful in practice, since it clarifies quickly the thread of execution as well as the behavior of the scheduler. In Java, every program with a Swing or AWT graphical user-interface is necessarily multithreaded, and so this is especially useful for visualizing the behavior of the AWT-Event thread in conjunction with other data processing threads. Figure 8.4 provides an example of a sequence diagram that uses colors to highlight different threads' behaviors.

It is possible, though uncommon, to show multiple, alternate paths of execution using UML sequence diagrams by splitting a lifeline and depicting the messages on the appropriate lifeline. This allows a diagram author to convey information about different yet related cases in one diagram. For example, alternate cases of a conditional statement can be shown in one diagram by splitting the lifeline. JIVE sequence diagrams, which represent actual program execution, cannot use such a nondeterministic branching: at any point in time in the execution sequence, exactly one instruction is being executed. The JIVE methodology treats program execution as an immutable sequence of events, and so there is no concept of nondeterminism in the execution log.

The split-lifeline approach can also be used to show multiple method activations on one lifeline, as occurs when an object calls a method on itself. This also occurs in cases where double-dispatch is used, as in the Visitor design pattern [32]. However, this approach to drawing uses a great deal of horizontal screen space. Given a lifeline that has n concurrently executing method activations, a horizontal space between methods of s, a method rectangle width of r, and an object width of w, the lifeline requires $max(w, (n-1)s + nr)$ horizontal space. Our solution to this problem is *method stacking*. Rather than splitting the lifeline, the method rectangles are placed adjacently, side-by-side, on the single lifeline. These two drawing techniques are shown in Figure 8.5, which clearly illustrates the space saving of method stacking. In the method stacking approach, the horizontal space needed by a lifeline is reduced to $max(w, nr)$. Although both use horizontal space that is linearly proportional to the number of method activations, the stacking approach uses less space by a factor of the minimum space between split lifelines. This is significant when optimizing the

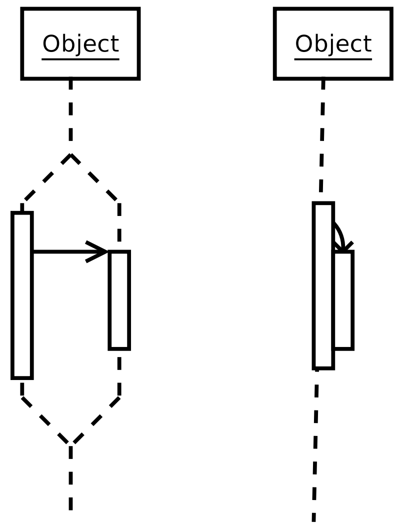


Figure 8.5: Two sequence diagram representations of the same data. The split-lifeline approach (figure left), potentially uses much more screen space than the stacked approach (figure right).

usage of screen space, where the constants that are dropped in an asymptotic analysis become the difference between fitting a drawing on the screen or not.

8.2 Aesthetic Preferences

When manually drawing a sequence diagram, the position of objects, methods, and messages is determined subjectively: the arrangement of information in the diagram depends on what the author wishes to convey, personal desire for artistic expression, and a set of generally-implicit aesthetic constraints. Many of these constraints are intuitive, such as preferring smaller diagrams over inordinately large ones. Ambler [5] quantified some of these human preferences, including a definite preference for a left-to-right ordering of methods according to their activation times.

In contrast to manual drawing, automatic drawing of sequence diagrams requires that aesthetic preferences be explicit and specific. We propose the following aesthetic constraints, based upon a survey of sequence diagrams and the extensive existing work on general graph-drawing aesthetics [24]:

1. *Tight arrangement of methods on a lifeline.* When there are multiple method activations on a lifeline at any point in time, the method rectangles should be arranged so that method rectangles take up minimum total area.

2. *Minimize edge length.* The length of method rectangles is determined by the amount of time the method was active and is therefore fixed. However, the length of message arrows is directly related to the distance between object lifelines, which is in turn determined by the horizontal

ordering of objects. As in general graph drawing, it is preferable to have shorter edges when possible.

3. *Minimized crossings between messages and methods.* It is undesirable for message arrows to cross over method rectangles. Again, the presence of absence of such crossings is determined by the horizontal ordering of objects. We do not consider the crossing of message arrows over lifelines without method activations since this is implicitly handled in our criterion of minimized edge lengths: shorter edge lengths will cross fewer lifelines, assuming constant width allocation for objects.

4. *Left-to-right ordering of messages.* As mentioned previously, existing work favors a left-to-right ordering of messages. This corresponds to the natural tendency to read rightwards as is common in many written natural languages. A leftward message call is a *back edge* of the sequence diagram.

5. *Preservation of object clusters.* When objects belong to a logical cluster, we prefer that these objects be adjacent in the sequence diagram. The definition of an object cluster depends on the environment being used to draw the sequence diagram. Clusters may be specified statically by the user, or they can be automatically determined through program analysis (see Chapter 9).

The first criterion is specific to the drawing of a single lifeline, while the others are directly related to the horizontal ordering of objects and, by extension, their lifelines. These dual problems of arranging methods on a lifeline and arranging the lifelines themselves are essentially separable, and so we address them as such. We examine the problem of method arrangement in Section 8.3 and the problem of lifeline arrangement in Section 8.4.

8.3 Arrangement of Methods

According to our method-stacking approach, when there are multiple methods concurrently active on one lifeline, the rectangles are arranged side-by-side along the lifeline. Recall that the vertical placement of a method rectangle is fixed, as it is based on the sequence data, whereas the horizontal placement is variable, as long as the method activation is associated with the proper object's lifeline. Within JIVE sequence diagrams, we depict this association by ensuring that concurrently execution methods are contiguously connected to their associated lifeline and no other lifelines.

The difficulty of the method stacking problem depends on whether the program execution is single-threaded or multi-threaded. In a single-threaded program, there is a single sequence of calls, and every method must return before its calling context is active again. In a multi-threaded program, this is not the case, since there can be multiple paths of execution, with the methods on separate threads effectively behaving independently with respect to their lifeline. We can prove the following property of single-threaded programs:

Property 8.9: Method Tightness. If a (Java) program is single-threaded, then all of the methods on a single lifeline can be drawn using the stacking paradigm in minimal horizontal space and maintaining left-to-right ordering of method invocations.

Proof. Let m_0, m_1,...,m_n be a sequence of method calls on an arbitrary object context, sorted chronologically, such that m_i calls m_{i+1} for all $0 \leq i < n$. Assign the method rectangles to bins so that m_i is in bin i for all $0 \leq i < n$; that is, use the stacking paradigm.

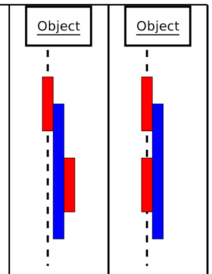
Assume that there is a method m_k that can share a bin with another method $m_{k'}$, where k and k' are in the sequence of method calls; assume without loss of generality that $k' > k$. This arrangement yield a tighter packing than the normal allocation of methods. However, the ending time of m_k must be after the starting time of $m_{k'}$, by definition, and hence placing these two methods in the same bin will yield a collision, an overlap of method rectangles. This collision would specifically stretch from $start(m_{k'})$ to $end(m_{k'})$ since $start(m_k) < start(m_{k'})$ and $end(m_k) > end(m_{k'})$. Therefore, our assumption is invalid, and by contradiction, the original bin allocation gives the tightest packing for the sequence of methods along a lifeline. $\square$

In the domain of multi-threaded programs, such a tight packing is not always possible, as can be shown with an example. For simplicity, we will consider only a single lifeline, which is sufficient to illustrate the case. Consider two threads, *Red* and *Blue*, and a method, m. Figure 8.6(a) is a sequence of method call and return messages with the thread indicated. Parts (b) and (c) of the figure show two different renderings of the sequence data. In Figure 8.6(b), the left-to-right ordering of methods (with respect to their time of activation) is maintained, but the area used by the drawing is not optimal. In Figure 8.6(c), the left-to-right ordering criterion is not met, but the area used by the diagram is optimal.

As the example demonstrates, the technique for drawing single-threaded programs does not scale seamlessly to multithreaded programs, given that we wish to minimize the area used for each lifeline. The two aesthetic criteria to balance are the left-to-right ordering of methods and the minimization of drawing area. However, the left-to-right ordering can be relaxed without losing expressiveness of the diagram: the order of method invocations is still represented in the vertical placement of

Time	Message	Thread
1	call m	Red
2	call m	Blue
3	return	Red
4	call m	Red
5	return	Red
6	return	Blue

(a)



(b) (c)

Figure 8.6: A series of messages in a multithreaded program, along with two renderings of the corresponding lifeline. Part (b) is a suboptimal layout in area, though it preserves left-to-right ordering of messages. Part (c) is an optimal layout in area, but it contains a backward call.

rectangles, and so the left-to-right ordering is inherently somewhat redundant. For example, given two method activations m_i and m_j, where $start(m_i) < start(m_j)$, then the method rectangle for m_i must start above the method rectangle for m_j, regardless of their horizontal ordering. The vertical placement is determined by sequence diagram semantics, whereas the horizontal placement is merely an aesthetic preference. With the corresponding constraint relaxed, we can focus on the area minimization of method rectangles on a lifeline.

The method arrangement problem itself can be specified as follows: given a sequence of n intervals $[s_1, e_1]$, $[s_2, e_2]$, …, $[s_n, e_n]$, where each s and e are unique positive integers, assign labels $L(s_i)$ for $1 \leq i \leq n$ such that, for any pair of intervals $[s_i, e_i]$ and $[s_j, e_j]$, if the intervals overlap, then $L(s_i) \neq L(s_j)$. The intervals correspond to the start (s) and end (e) times of method activations. The uniqueness clause is a property of our data: in a uniprocessing environment, only one event can occur at any discrete point in time, and therefore all messages have a unique integer index. The sequence of intervals can be extracted from a sequence graph representation using Algorithm 5.2.

Our solution to the method arrangement problem is to allocate "bins" to each lifeline, where there is one bin per label, and the labels are therefore equivalent to horizontal bin indices. A bin is *available* at a time t if there is no method in the bin whose interval contains t. When a method must be placed on a lifeline, it is placed in the leftmost available bin, if one exists; otherwise, a new bin is added to the right, and the method is assigned to it. The time taken by this approach is proportional to the product of the number of messages in the sequence and the maximum number of simultaneous method calls, which is, directly, the maximum label assigned (*i.e.* the index of the rightmost bin). This approach is presented as Algorithm 8.1.

Algorithm 8.1 takes $O(n^2)$ in the worst case, where n is the number of intervals. The worst case occurs when $s_i < s_{i+1}$ and $e_i > e_{i+1}$ for all $0 \leq i < n$, as happens when a single-threaded program

Input: Sequence S of n intervals $[s_0, e_0], [s_1, e_1], \ldots, [s_{n-1}, e_{n-1}]$
Output: Labels $L(i)$ for each interval i in the input sequence

```
//L will contain the result.
L ← empty map;
//M will map from bin number to the interval it contains.
M ← empty map;
rightmost ← 0;
foreach Interval I = [s_i, e_i] in S do
    //The timeslice currently under consideration.
    now ← s_i;
    //Find the leftmost available bin.
    leftmost ← null;
    j ← 0;
    if rightmost = 0 then
        leftmost ← 0;
    else
        while j < rightmost do
            if M(j) ≠ null then
                if end(M(j)) < now then
                    //Method in bin j has ended.
                    M(j) ← null;
                    if leftmost ≠ null then
                        leftmost ← j;
                    end
                end
            else
                if leftmost ≠ null then
                    leftmost ← j;
                end
            end
            j++;
        end
    end
    //Assign this interval to the leftmost bin and update local map.
    L(i) ← leftmost;
    M(leftmost) ← I;
    //Update the rightmost pointer if necessary.
    rightmost ← max(leftmost+1, rightmost);
end
return L
```

Algorithm 8.1: ArrangeMethods: Arrange method rectangles on a lifeline.

consecutively executes n methods on the same context. In this case, the number of iterations through the loop at line 11 approaches n. However, in practice, there are usually few method activations on any context, and the number of times through this critical loop is quite low. A clear exception is a recursive method that continually invokes on the same context (as opposed to a recursive method that is invoked across an object structure, such as a search through a linked list). The algorithm as presented is for a single lifeline. In order to place the methods of a sequence diagram, it must be applied to all lifelines. The time taken to do this is quadratic in the number of method activations, although reasonable execution speeds are expected for most programs, as described above.

8.4 Arrangement of Objects and Lifelines

In this section, we explore the problem of horizontal arrangement of objects in the sequence diagrams so as to satisfy the proposed aesthetic preferences. Unfortunately, the problem of determining a horizontal ordering of objects that minimizes even just the edge lengths is NP-hard since we can reduce the the *Minimum Linear Arrangement (MinLA)* problem to it. The MinLA problem is, given a graph K, to assign a unique integer label $L(v)$ for each node v of K such that $\sum_{(u,v)\in E_K} |L(u) - L(v)|$ is minimized, where E_K is the set of edges of K [34]. This problem is equivalent to the horizontal lifeline arrangement problem if we consider each object to be a vertex and the messages among objects to be edges.

Simulated Annealing Approach

Since the minimization of edge lengths is NP-hard, an efficient technique is needed for drawing sequence diagrams. *Simulated annealing* is a probabilistic meta-algorithm for global optimization problems, independently developed by Kirkpatrick et al. [59] and Cerny [19]. The name comes from the metallurgical annealing process whereby crystals are heated and then their cooling is controlled so as to increase the crystals' sizes and reduce their defects. Heating causes atoms to “wander” from their states of local minima, and slow, controlled cooling facilitates their finding optimal states. Simulated annealing then uses probabilistic mathematical models to simulate the annealing process, forcing variables out of local, suboptimal states in hopes that on termination, they find stability in optimal states. Experimental studies have shown that simulated annealing can give reasonable solutions to MinLA [71]. Inspired by this, we have designed a simulated annealing program to compute a left-to-right order of the objects that optimizes the aesthetic criteria. The program is similar to the one used for solving MinLA.

The simulated annealing approach does not require any information on threads or the length of methods, and hence we embed a sequence graph as a directed graph, following the methodology of Algorithm 5.1. In fact, it is sufficient to only consider method calls, not method returns: the number of method calls equals the number of method returns in a complete sequence diagram generated by the JIVE methodology, and so considering both is redundant. We ignore exceptional returns for simplicity's sake; in practice, we treat exceptional returns the same as normal returns in the diagram. Let $G = (V, E)$ then be a directed graph whose vertices are lifelines such that there is a directed edge $e = (u, v)$ in $E(G)$ if and only if there is a method call from u to v, where u and v are nodes of $V(G)$ and $u \neq v$; those cases where $u = v$ have no impact on the arrangement of lifelines. Associate with each edge $e = (u, v)$ of $E(G)$ a weight $w(e)$ whose value is equal to the number of calls made from lifeline u to lifeline v throughout the recorded program's execution. Note that $w(e)$ will actually be half the number of edges between lifelines u and v, since we are only considering method calls for the moment. Let A, B, and C be three user-defined numbers that represent respectively (A) the relative importances of minimizing the edge lengths, (B) the number of back edges, and (C) the number of crossings between the message edges and method rectangles. Let $0 < \alpha < 1$ be the *cooling factor*, a user-specified number that is necessary in simulated annealing to represent the relative speed with which the system seeks stability. Our optimization goal is to assign a unique integer label $L(v)$ to each node v of G such that the objective function F is minimized, with F is defined as

$$\begin{aligned} F = A \cdot & \sum_{(u,v) \in E} \{w((u,v)) \cdot |L(u) - L(v)|\} \\ + B \cdot & \sum_{(u,v) \in E} \{w((u,v)) \cdot [L(u) > L(v)]\} \\ + C \cdot & \sum_{(m_1,m_2) \in E, m_3 \in V} [(m_1, m_2) \text{ } \mathit{crosses} \text{ } m_3], \end{aligned}$$

where x *crosses* y is equal to 1 if the message edge x crosses method rectangle y, and otherwise it is equal to 0. Here, E is the set of the edges and V is the set of vertices of G. Let t be the *initial temperature*, a variable whose initial value is assigned by the user. Then a program for horizontal arrangement of objects using simulated annealing is given as Algorithm 8.2. The efficiency of this algorithm depends on α and the initial value of t. In an experimental system, these can be specified by the user.

The simulated annealing approach will produce good drawings of sequence diagrams but only

Input: Sequence graph G, Objective function F, Initial temperature t, Cooling factor α, Minimum temperature m
Output: Labels for each object lifeline

repeat
    $u \leftarrow$ random vertex of G;
    $v \leftarrow$ random vertex of G;
    compute F assuming labels of u and v are swapped;
    $\delta \leftarrow$ (value of F with exchange) $-$ (current value of F);
    With probability $\min(1, exp(-\delta/t))$, swap the labels of u and v;
    $t \leftarrow \alpha t$;
until $t \leq m$;

Algorithm 8.2: AnnealingArrangement: Use simulated annealing to horizontally order object lifelines.

if appropriate values of A, B, C, α, and the initial t are supplied. Good default values can be determined by experimentation, but the simulated annealing approach still depends heavily on user preferences or intervention. Furthermore, it is generally the case that the longer a simulated annealing program runs, the better the probability of that it will produce good results; this is a complication of using a probabilistic approach. However, it is often not feasible within an interactive execution system to allow the simulation to run indefinitely. Even pushing this computation onto a dedicated thread or process is undesirable since objects in the lifeline will acquire initial placements, and it would be to the user's detriment to move them once placed. It is possible to add another term to the objective function F, one whose factor would represent the relative importance of maintaining a certain arrangement; this could prevent major changes in the lifeline placements, but it could also then hinder other aesthetic goals and the finding of a good solution. Hence, although simulated annealing can produce good drawings, especially when produced offline (*i.e.* not in the context of interactive visualization), it is desirable to have alternatives.

First-Come First-Served Approach

Given a static sequence model, a good drawing can be generated using the simulated annealing approach described above. However, in an environment where the sequence data is being generated, and there is a need to display partial information, then it is not clear that the previous approach is feasible. An alternate methodology for arranging lifelines is the *first-come first-served approach*, which finds fast solutions occasionally at the expense of ignoring several aesthetic criteria. This approach is very simple and is defined by the following rule: for any objects u and v, if u is created after v, then $L(u) > L(v)$. That is, the lifelines are given a partial ordering according to their creation times. There is no constraint for lifelines that are created at effectively the same time,

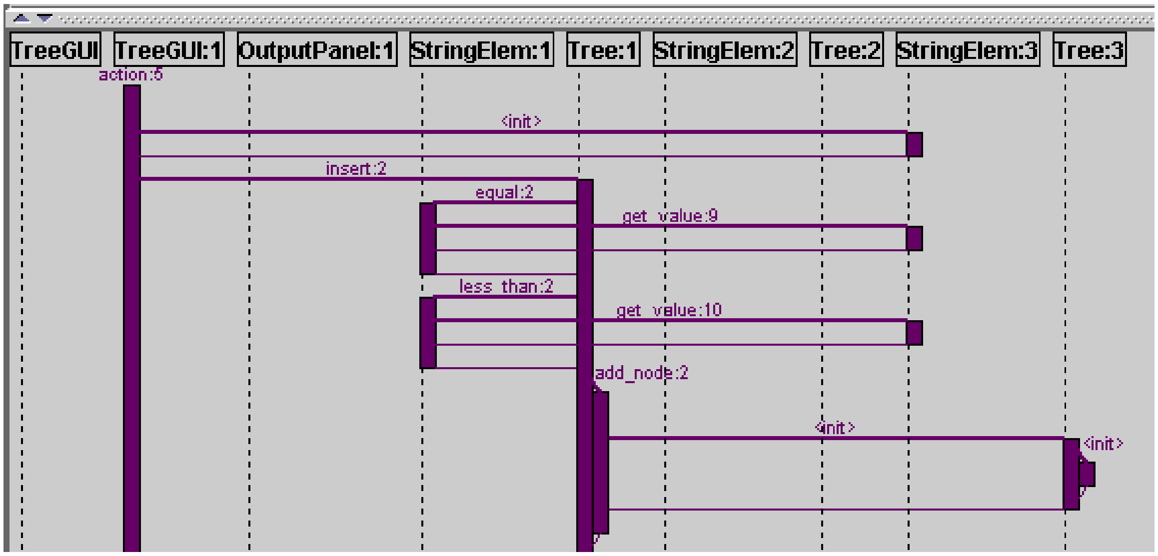


Figure 8.7: A sequence diagram that was generated by JIVE using the first-come first-served approach. A better drawing, according to the aesthetics described in Section 8.2, would have reversed the positions of `StringElem:1` and `Tree:1` to reduce the number of back edges.

for example, the static contexts loaded at program initialization. This approach is convenient for drawing sequence diagrams during interactive execution, such as is possible with JIVE. When a new lifeline is created, its position in the sequence diagram can be determined in constant time. During offline analysis, it takes only $O(m)$ time to arrange the lifelines, where m is the number of sequence events and these events are already sorted. Although this approach ignores several of our drawing aesthetics, it does guarantee that object lifelines are ordered by creation time, which is a useful visual cue to the reader. This matches the approach by which sequence diagrams are generally constructed during program design [15, 44]. That is, we strictly apply the preference for left-to-right messages it to the lifeline creation messages, which a first-come first-served ordering.

The major disadvantage of this approach is that it does not explicitly address aesthetic criteria associated with lifeline arrangement as presented in Section 8.2. An example of this effect is given in Figure 8.7. The figure shows a screenshot of an application that builds trees, where each tree node is a `Tree` object that references a `StringElem` object. The `StringElem` objects each hold a Java `String` data item. In the figure, a first-come first-served approach is used, and so the sequence of lifelines reflects the order in which objects were created. However, `Tree:1` has two back-edges to `StringElem:1`, namely the calls `equal:2` and `less_than:2`. These two back-edges could be eliminated by reversing the positions of `StringElem:1` and `Tree:1`, an operation that would also reduce the total edge length of the diagram. As presented, the diagram was drawn using the first-come first-served approach, which does not take these back-edges into consideration.

Summary and Observations

The approach used to draw sequence diagrams depends upon whether the sequence data is entirely available at the time of drawing or if it is being produced asynchronously. In the former case, there is a good argument for using simulated annealing to arrange lifelines, since this approach is known to work well for the minimum linear arrangement problem. However, JIVE generates sequence data during program execution, and so it uses the first-come first-served approach due to its speed and automation (*i.e.* no user interaction or preferences are required). In either case, the arrangement of methods on each lifeline is a separate problem, and the approach of Algorithm 8.1. Algorithm 8.3 is the approach used in JIVE to draw sequence diagrams. Note that in the JIVE methodology, sequence data is never removed, and a sequence diagram's size increases monotonically. Neither the simulated annealing nor the first-come first-served approach satisfy all of our aesthetic and efficiency requirements, and so the latter approach is used based on its simplicity and speed. In the next chapter, we examine how more advanced clustering techniques can be used to improve the drawing of sequence diagrams while maintaining their relationship with the program source code and the object diagram structure.

Input: Sequence data S;
Minimum spacing between lifelines m;
Height of a time segment h;
Width of method activation w;
Maximum y-coordinate Y
Output: Sequence diagram

$Q \leftarrow$ empty sequence;
if *S represents complete sequence information* **then**
    Arrange the lifelines of S using AnnealingArrangement (Algorithm 8.2); record this ordering in Q;
else
    Sort the lifelines of S according to object creation time (*i.e.* use the first-come first-served technique); record this ordering in Q;
end
//We assume that a lifeline is encoded as a sequence of intervals that can be fed to the ArrangeMethods algorithm subroutine.
foreach *Lifeline L_i of S* **do**
    $M \leftarrow$ ArrangeMethods(L_i);
    $maxM \leftarrow 0$;
    foreach *Method m in L_i* **do**
        Draw a method rectangle of width w from $h * start(m)$ to $min(Y, h * (end(m) - start(m)))$ at column $h * M(m)$;
        $maxM \leftarrow max(maxM, M(m))$;
    end
    //Record the width of the lifeline's method activations.
    $W_i \leftarrow h * maxM$;
end
$x \leftarrow 0$;
foreach $L_i \in Q$ **do**
    Let W be the width of the object at the top of L_i;
    Draw lifeline L_i at column x;
    $x \leftarrow x + max(W, W_i) + h$;
end

Algorithm 8.3: DrawSequenceDiagram: Draw a sequence diagram.

Chapter 9

Towards Improved Object and Sequence Diagrams

The techniques presented in the previous chapters adopted a primarily graph-theoretic approach to drawing object and sequence diagrams. Our aesthetic constraints are derived from the specific drawing domain, and the resulting drawings were reasonably good. However, these techniques can be improved, especially with respect to the preference that drawings somehow reflect the semantics of their program. In order to address this, we observe that the drawings generated by JIVE are not arbitrary graphs, but representations of program execution. Furthermore, program execution is directly related to a program's definition. Hence, we can use properties of a program's definition, its static structure, in order to build better visualizations of the program's execution, its dynamic behavior.

Good software engineering practice follows *use-case driven design*. In this development model, defining and analyzing use cases is the first step. Classes, relationships, and interactions emerge from this analysis of the use cases. An good application architecture is therefore well structured, and this structure can be exploited in our drawing techniques. Poorly structured applications may lead to an undisciplined, tangled mess in the object graph, and in such degenerate cases, the only recourse is to use standard drawing techniques and hope for a serendipitous correspondence between the architecture and the object structure. By contrast, well-structured applications should yield predictable object structures. It is this observation that underlies our technique for improved drawing techniques for object and sequence diagrams.

In this chapter, we present techniques for detecting and predicting object *clusters*. A cluster is a group of nodes in a graph that are to be drawn in close proximity to each other. There are several techniques for drawing clusters; for example, `dot` allows the user to define clusters on the input, and these are drawn within a bounding box in the drawing [33]. This approach produces a layered

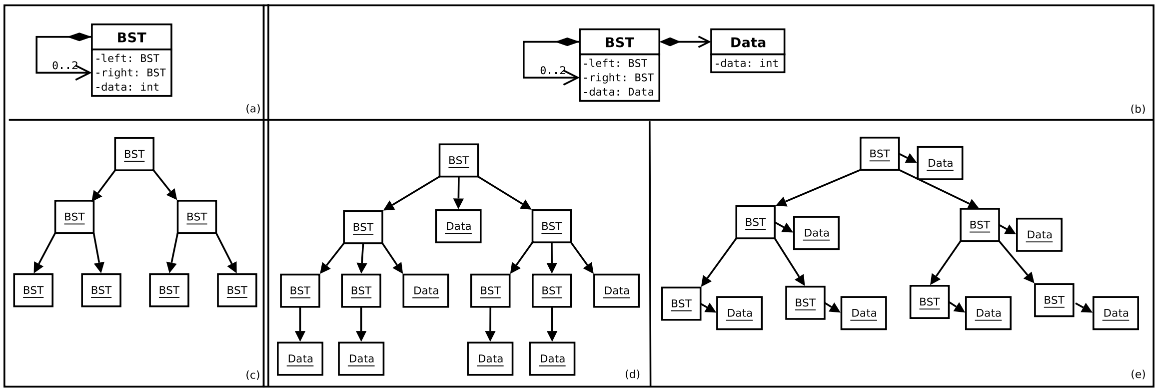


Figure 9.1: Binary search trees. Parts (a) and (b) show two alternate implementations for a binary search tree. Part (c) is an object diagram for the architecture in (a), and parts (d) and (e) show two possible object diagrams for the architecture in (b). Object diagrams (c) and (d) were drawn using standard layered graph drawing techniques and aesthetics. Diagram (e) was drawn by incorporating an analysis of the class diagram into the drawing.

drawing, but clusters can also be defined in force-directed drawings. The clustering algorithms we define can be used in any situation where clustered drawing is possible. In JIVE object diagrams, we use a layered drawing strategy, but the clusters are also used in JIVE sequence diagrams.

A good example motivating the need for class diagram analysis in drawing object diagrams is provided by the binary search tree example. Figure 9.1 shows two designs for a binary search tree along with sample object diagrams. Figure 9.1(a) shows a class diagram for an implementation of integer binary search trees that encapsulates integer values within the tree nodes. This is a reasonable design in a language such as Java, where integers are represented with a primitive type rather than an object type. Figure 9.1(b) shows an alternate design that stores the value of a node in a separate object. Figure 9.1(c) shows a sample object structure for design (a), and parts (d) and (e) show two drawings of a sample object structure for design (b). The drawings in parts (c) and (d) have been generated using standard layered graph drawing techniques and aesthetics: topologically sort the nodes, assign them to layers, and reduce the crossings [24]. Diagram (c) emphasizes the hierarchical structure of the tree: each layer in the drawing represents a logical layer in the tree data structure. This highlighting of hierarchical structure is one of the benefits of layered graph drawing [85]. However, the drawing of part (d) lacks the clarity of (c). The underlying meaning of the structure is harder to determine from the visualization since `Data` and `BST` objects are treated homogeneously by the drawing algorithm. A better drawing of the same structure is found in part (e). This drawing emphasizes the tree structure, keeping each logical layer on a visual layer in the drawing, while the `Data` objects are more closely tied to their corresponding `BST` objects. This coupling in the drawing reflects the tight coupling of these two classes in the design, especially the topological dependency of `Data` objects upon their `BST` counterparts, where each `Data` is referenced by exactly one `BST`.

The remainder of this chapter is organized as follows:

- Section 9.1 describes the extended aesthetic criteria involved in this approach; these are discussed as an extension to the aesthetic criteria of object and sequence diagrams as presented in the previous two chapters.
- Section 9.2 presents an overview of our class diagram analysis technique, focusing specifically on the nature of binary class relationships.
- Section 9.3 presents our technique for detecting and predicting object clusters.
- Section 9.4 presents our algorithms for detecting clusters using them for enhanced drawing of object and sequence diagrams.
- Section 9.5 provides a small case study illustrating the operation of our algorithms on a program that a student might write for an introductory computer science course.

9.1 Aesthetic Criteria

Effective graph drawing is achieved by balancing multiple conflicting visual aesthetic constraints. Managing these constraints is a combinatorially hard problem. We augment basic graph-drawing aesthetics, such as minimized total edge-length and minimized area, with aesthetic criteria that are specific to object diagrams. The overarching aesthetic goal is that related objects are drawn in proximity. Since "related objects" is vague, we specify the following concrete properties we wish to capture in our drawing:

A1 *Couple leaf objects with their aggregators.* We consider *leaf objects* to be those objects that do not reference other objects, and that are themselves referenced by exactly one *aggregator object*. We expect there to be a large number of leaf objects in an arbitrary program due to the power law distribution of object references: in normal programs, there are very few objects with a large number of references and there are very many objects with a small number of references [73]. This criterion is proposed as a technique to reflect the tight coupling of simple objects in program design with proximity of vertices in an object diagram.

A2 *Cluster recursive object structures.* The binary search tree of Figure 9.1 is a classic example of a recursive type: the tree is defined by repeated linking of objects of type `BST`. Recursive structures are widely used in programming as data structures, and their use is encouraged by several design patterns such as Composite and Decorator [32]. This criterion is proposed as a technique for highlighting the cohesiveness of a recursive structure. The intention is that objects that make up a recursive structure will be clustered into a topologically-constrained space of the drawing, which highlights their structural cohesiveness and conceptual grouping.

We propose that these aesthetic criteria will yield object diagrams that more closely match the architectural specification of the class diagram. In this document, we show that these properties can be detected and utilized in drawing by combining static class diagram analysis with dynamic analysis of object diagrams.

9.2 Class Diagram Analysis

Class diagrams are an important tool for software engineering. A class diagram is a visual representation of the static architecture of a system; dynamic behavior is not explicitly shown in a class diagram and is reserved for interaction diagrams. We adopt the Unified Modeling Language notation for class diagrams. However, because the UML is a general modeling language, there is not necessarily a precise translation from UML class diagrams to any particular programming language syntax. Many tools exist that correlate class diagrams and source code, but it is important to realize that the expressiveness of the UML's notation transcends any specific programming language. In this section, we explore class diagrams, their relationship to object diagrams, and the inherent correlations that can be exploited to produce better automatic drawings of enhanced object and sequence diagrams.

Binary class relationships are the relationships between pairs of classes within a class diagram. The commonly used binary class relationships are generalization, realization, dependency, association, aggregation, and composition. Generalization and realization are both used to represent inheritance, the former for implementation inheritance and the latter for interface inheritance. It is important to note that, while Java explicitly differentiates between *classes* and *interfaces*, UML sequence diagrams use classes for both, annotating interfaces with the ≪interface≫ stereotype. In Java, `extends` is used for classes (implementation inheritance) and `implements` is used for interfaces (interface inheritance).

The analysis of dependency, association, aggregation, and composition relationships is complicated by their ambiguity with respect to operational semantics. In a sense, there is a discontinuity between the model of a system and its implementation unless the model is the program itself, as in visual programming languages. The only clear way to uniquely map from a UML model to source code is to place restrictions on the modeling language, as described by Harrison et al. [47]. Rather than narrowing the modeling language or accepting the ambiguities, we adopt the consensual definitions of binary class relationships as proposed by Guéhéneuc and Albin-Amiot [42]. The consensual definitions for association, aggregation, and composition, as they relate to model semantics, are summarized below.

Dependency A dependency from A to B implies that there is a sense in which instances of B will depend on instances of A, but there is no semantic implication.

Association An association from A to B in a model means that instances of A can directly send messages to instances of B. This implies that the instance of A must have a reference to an instance of B either through a field or a method's local variable.

Aggregation An aggregation is an association from A to B and is a relationship between a *whole* and its *part(s)*, respectively. This implies that the instance of A has a reference to an instance of B through a field or possibly abstracted through a container class.

Composition A composition as an aggregation from A to B such that the instances of B are exclusively referenced by their corresponding instance of A. Furthermore, there is a lifetime dependency of B with respect to A such that if an instance of A is destroyed, its corresponding B objects are likewise destroyed. This implies that an instance of A has a reference to an instance of B through a field (or through a container class), but no other object has a reference to that same instance.

These definitions assume that each binary class relationship is directed, and this assumption is safe since any bidirectional relationship can be expressed as a pair of directed relationships [61]. Dependency, association, aggregation, and composition form a hierarchical ontology of relationships: compositions are aggregations, aggregations are associations, and associations are dependencies. However, there is no direct correlation between dependencies and source code, and hence we focus our analysis on the other three relationships. Aggregations and compositions are inherited through generalization except when prohibited by access control modifiers, overriding, or shadowing. The reason for this inheritance is that associations, aggregations, and compositions are reified as field references, and fields are inherited through subclassing, except where explicitly prohibited. However, these references may include some levels of indirection through container classes, but this is addressed later.

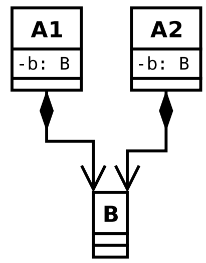


Figure 9.2: A class diagram showing two compositions to the same class. $A1$ and $A2$ are the "wholes" and B is the "part" in the compositions.

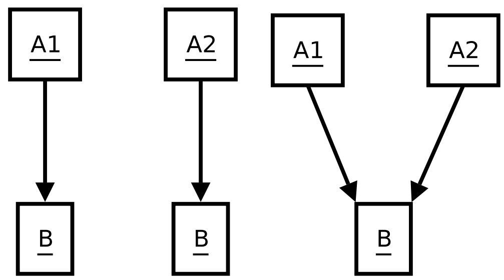


Figure 9.3: Two object diagrams illustrating (a) legal and (b) illegal realization of the composition of Figure 9.2 in object diagrams. Part (a) is legal since the `B` objects are exclusively referenced, and this is a requirement imposed by the semantics of composition. Part (b) is illegal since the lone `B` instance is non-exclusively referenced.

The presence of any association, aggregation, or composition does not preclude the existence of other relationships. The exclusivity of a composition refers to the object level, not the class level. An example is shown in Figure 9.2; the fact that the relationships are compositions implies that the B object referenced by an instance of $A1$ will be different from the B object referenced by an instance of $A2$. In fact, this is a generalization of the property that the B object referenced by any $A1$ object must be different from the B object referenced by any other object. Figure 9.3 provides a legal and an illegal object diagram for the class diagram of Figure 9.2. It is interesting to note that Figure 9.3(b) is illegal despite its symmetric resemblance to its class diagram. Hence, structural symmetries in a class diagram do not necessarily imply similar symmetries in the object diagram; it is necessary to take into account the precise semantics of the relationships themselves, not just the graph-structural properties of the class diagram.

Our analysis requires looking beyond binary class relationships to how these relationships are composed in class diagrams. To this end, we define two categories of relationship paths: generalization paths and association paths.

Definition 9.4. A **generalization path** is a sequence of class identifiers $\tau_1, \tau_2, \ldots, \tau_n$ such that

τ_{i+1} is an immediate superclass of τ_i for all .

Definition 9.5. An **association path** in a class diagram is a sequence of class identifiers $\tau_1, \tau_2, \ldots, \tau_n$ such that for any pair of types (τ_i, τ_{i+1}), $1 \leq i \leq n-1$, either:

- there is an association from τ_i to τ_{i+1}, or
- there is a generalization path from τ_i to τ_i', there is a generalization path from τ_{i+1} to τ_{i+1}', and there is an association from τ_i' to τ_{i+1}'.

Association paths are defined in terms of generalization paths since associations are inherited by subclasses. An association path is generic: it can contain aggregations and compositions as well as "normal" associations. *Aggregation paths* and *composition paths* are defined similarly to association paths, substituting aggregations and compositions respectively.

Binary relationships in object diagrams

The relationships between classes in a class diagram are realized as connections between objects at runtime. These connections are the structural, static, and method return links of the contour model as presented in Chapter 3. We proceed by analyzing each type of binary class relationship and explaining how they are realized at runtime.

Generalization and Realization. At runtime, inheritance hierarchies "collapse" into objects. It is possible to visualize the object in terms of its inheritance hierarchy [53], but generally the object can be treated as a single unit. Dynamic type checking will still recognize the object as inheriting from all of its superclasses.

Association. In order for an instance of A to send a message to an instance of B, it needs to obtain a reference to it. However, the presence of an attribute of type B in A would indicate an aggregation, according to the consensual definition. The reference to B must therefore be through a parameter to a method on A. The object reference from A to B is therefore a transient one, lasting at most as long as the method is active.

Aggregation. The implementation strategy for a part-to-whole relationship from B to A is to define attributes of type B within A [42]. This implies that in the object graph, instances of A will reference instances of B. Aggregation is not an exclusive relationship: these instances may be shared among other aggregate structures or may be bidirectional.

Composition Composition is an aggregation, and so it is also implemented through object attributes. The lifetime dependency of composition relationships is a constraint on the behavior of the object graph, specifically that the part is exclusively referenced by the whole. Given a class A that composes B and objects a of A and b of B such that a references b, then a is the only object that references b, and when a is destroyed, b is also destroyed.

Multiplicity and Recursive Types

The multiplicity of a relationship in a class diagram has a direct impact on the possible object structures that the program can generate at runtime. By definition, given a relationship from A to B where the multiplicity for A is m and the multiplicity for B is n, then at runtime, there can be up to m instances of A in relationship with B objects, and n instances of B in relationship with A objects. We refer to these cases as *bounded multiplicity*. The asterisk (*) is used by convention to indicate *unbounded multiplicity*, where there is no theoretical maximum to the number of objects in the relationship. Since we are analyzing arbitrary states of program execution, we treat the multiplicity as a potential maximum of objects in relation, not a predicted number. When a program begins execution, for example, these relationships will have zero objects. In the dynamic environment of program execution, it is therefore sufficient to treat any multiplicity k as if it were $0..k$.

The multiplicity of aggregations in a class diagram also provides important hints about the implementation of the aggregation. In general, given an aggregation from A to B with multiplicity of $k \in \mathbf{N}$, this is implemented in A as either k attributes of type B or as an array of B with size k. Given an aggregation from A to B with unbounded multiplicity, the implementation must be through either a dynamically-sized array or a collection class. Figure 9.4 gives a simple example in Java: part (a) shows an association with unbounded multiplicity; part (b) gives another view of the class diagram, this one closer to the implementation, highlighting the use of a `LinkedList` to realize the aggregation; and part (c) shows a sample object diagram for the aggregation. In the implementation of the `LinkedList` class, most likely there is a node class, such that the linked list composes nodes, and the nodes aggregate data objects (`B`, in our example). However, these are hidden in the class diagram, and so we have elided them from our object graph, as described previously.

The importance of recursive types in structured programming is evidenced by the many occurrences of recursive types in design patterns, including Decorator, Chain of Responsibility, and Interpreter patterns [32], in addition to their being standard fare in courses on data structures. Figure 9.5 shows an example of a recursive type, an instance of the Composite pattern, taken from the AWT package of the Java standard API. `Container` is a subclass of `Component`, but a `Container`

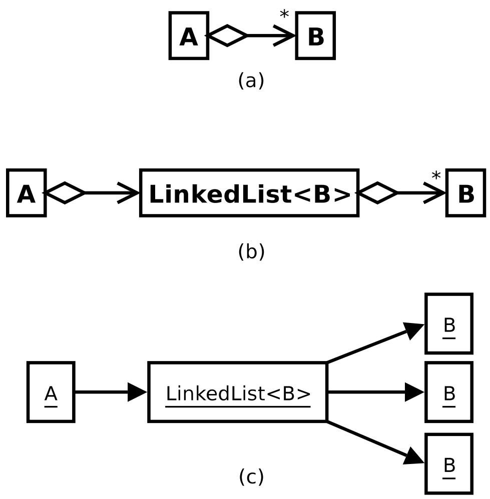


Figure 9.4: Aggregation realized as a linked list. Part (a) is a high-level class diagram, and part (b) shows how the aggregation in (a) is realized as a linked list in the implementation. Part (c) is a sample object diagram for the system.

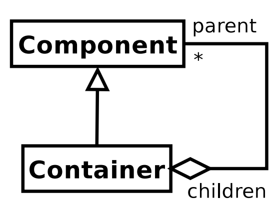


Figure 9.5: Sample recursive type from the `java.awt` package. The AWT/Swing API exemplifies recursive data types through the Composite design pattern.

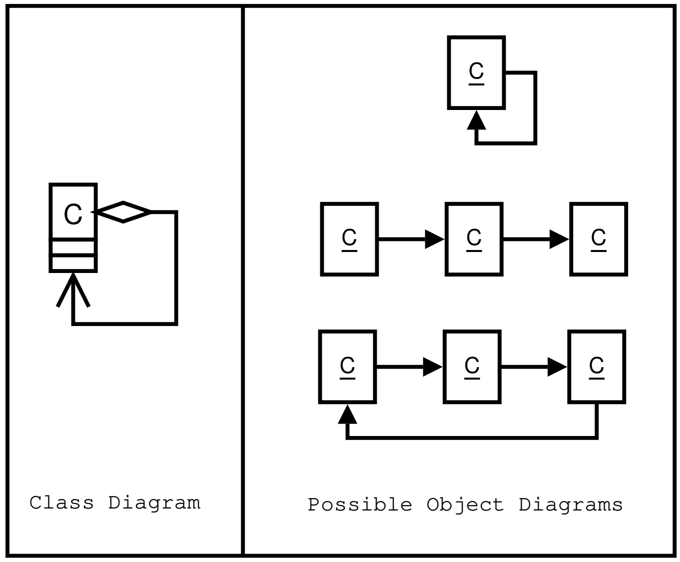


Figure 9.6: A simple recursive class and several possible object diagrams realizing the one-to-one self-aggregation.

object aggregates `Component` objects. These classes and their subclasses are used to build graphical user interfaces. At runtime, the object structure is a tree that is rooted in the main application window and whose branches are made up of more `Component` subclasses such as `Label`, `TextField`, and `Button` objects.

The simplest recursive types are self-aggregating classes, that is, those classes C that have an aggregation to C. A classic example is the basic binary search tree, as shown in Figure 9.1. Those recursive types that aggregate their superclasses exhibit similar behavior to those that are strictly self-aggregating; this is the case in the `Component-Container` example of Figure 9.5. This leads to our definition of a *simple recursive class*:

Definition 9.6. A **simple recursive class** is a class C that has an aggregation relationship with itself or with one of its superclasses. This aggregation is the class' **recursive aggregation**.

Given a simple recursive type whose recursive aggregation has a multiplicity of one, there are three archetypical realizations of the aggregation in the object diagram, as shown in Figure 9.6 with example class C. One possibility is that there will be only one instance of C, and it will bear a structural link to itself; in such a case, there is no multi-object structure to consider. Another possibility is that there will be multiple instances of C, and they will be linked by a series of directed object references. This type of disciplined linking is the basis of the linked list data structure. A third possibility is that there may be a chain of objects but with a single "back edge"; this forms a circular linked list structure. These examples demonstrate that simple recursive types with multiplicity of

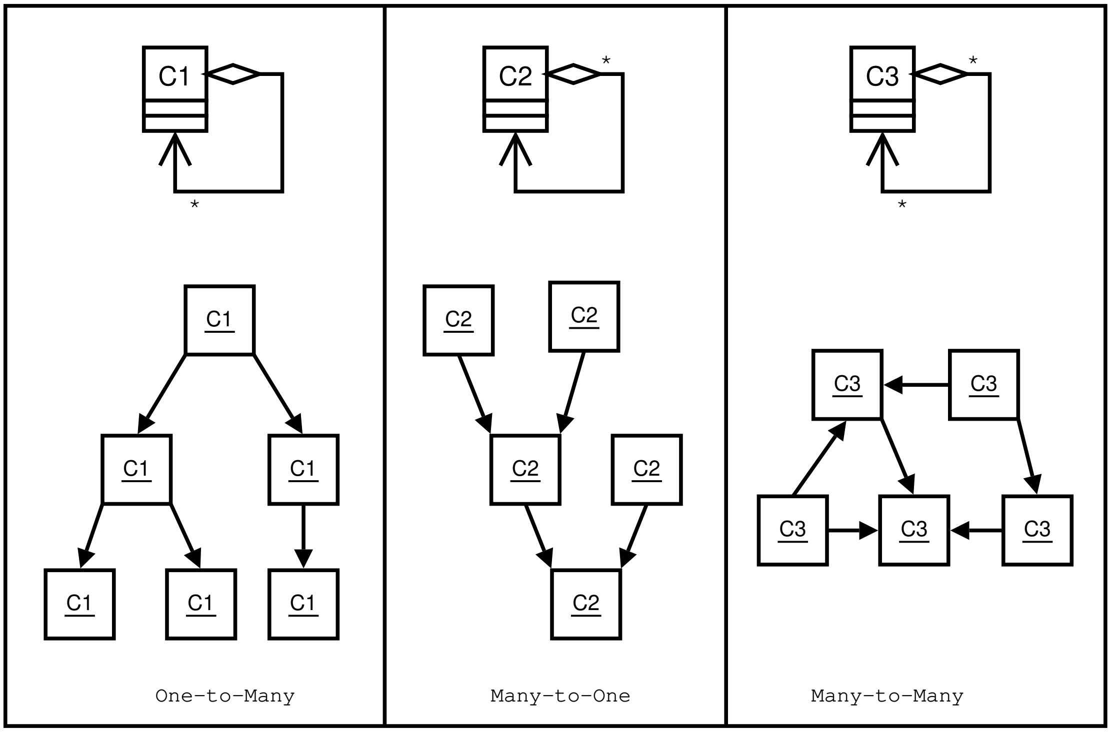


Figure 9.7: Class diagrams illustrating high multiplicity in a self-aggregation along with sample object diagrams.

one engender well-defined object structures, since in the worst case, there is only a chain of connected objects to consider, and in the simplest case, there is only one object.

It is worth noting that changing the multiplicity of the self-aggregation has a dramatic effect on the object diagram. We will examine three cases, examples of which are given in Figure 9.7. If the one-to-one relationship is replaced by a one-to-many relationship, represented as multiplicity "1..*" in UML, then each instance of `C` can potentially reference any number of other `C` objects, although each object is referenced by (at most) one `C` object. This forms the basis of tree-like data structures. If the self-aggregation is many-to-one, "*..1" in UML, then each instance of `C` references (at most) one `C` object, but each can potentially be referenced by any number of `C` objects. This forms an inverted tree, or a tributary-like data structure. Finally, in the case of many-to-many relationships, there can be any number of references among instances of `C`, and the resulting object structure is a general directed graph.

Mutually recursive types involve a cyclic aggregation path among classes in separate inheritance hierarchies. When restricted to aggregations of multiplicity of one, the behavior of mutually recursive types is similar to that of simple recursive ones. An example of a mutually recursive type is given in Figure 9.8. In this example, A and B are classes stand in a bidirectional aggregation, shown in

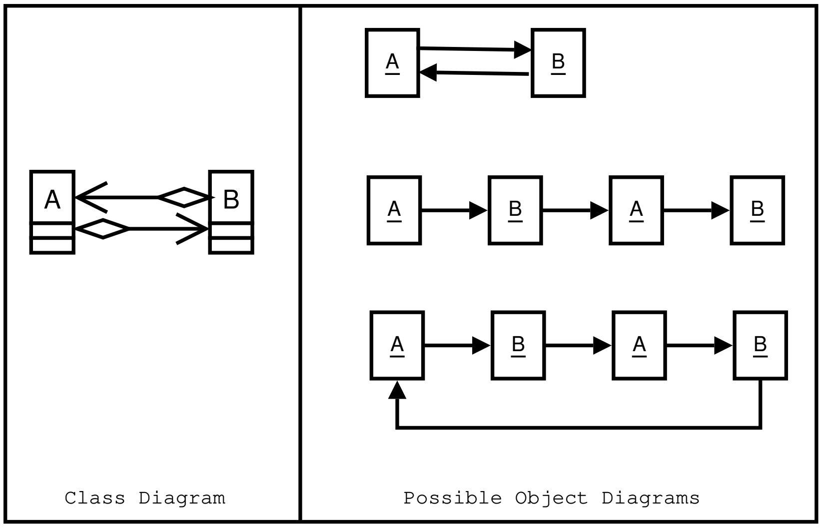


Figure 9.8: A class diagram for mutually recursive types `A` and `B`, along with some possible object diagrams.

the figure as two unidirectional aggregations for clarity. Part (a) is the class diagram, and parts (b), (c), and (d) show possible manifestations of the association as object references.

For clarity when discussing recursive classes, we can define a general *recursive type*, but we first must formalize a *generalized aggregation path*:

Definition 9.7. A **generalized aggregation path** is a sequence of class identifiers $c_1, c_2, \ldots, c_n$ such that for any subsequence $c_i, c_{i+1}, \ldots, c_j$, one of the following is true:

- $i - j = 1$, and c_i aggregates c_j.
- c_{i+1} is an immediate superclass of c_i, and there is a generalized aggregation path from c_{i+1} to c_j.
- c_{j-1} is a superclass of c_j, and there is a generalized aggregation path from c_i to c_{j-1}.

Definition 9.8. A **recursive type** τ is made up of a set S of classes such that either:

- $|S| = 1$ and the class $C \in S$ is simple recursive.
- $|S| > 1$ and for any pair of classes $C_1, C_2 \in S$, there is a generalized aggregation path between them.

A generalized aggregation path is an aggregation path that traverses generalization relationships. A generalized aggregation path from C to D can manifest in the object graph as a path of edges

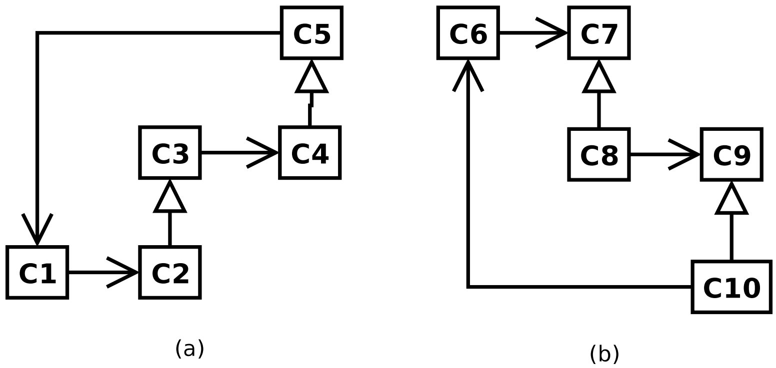


Figure 9.9: A pair of class diagrams containing cycles. Part (a) exhibits recursive types: classes `C1`, `C2`, and `C4` are all necessarily part of a recursive structure. Part (b) shows a similar structure, but it does not necessarily contain recursive types.

from an object of C to an object of D. The definition of a recursive type then is a formalism of the informal description provided above. We refer to any class diagram containing a recursive type as being a *recursive class diagram*.

Figure 9.9(a) provides an example recursive type that involves both generalization and aggregation. Given an instance of `C1`, it references an instance of `C2`, which inherits a reference to an instance of `C4`, which inherits a reference to an instance of `C1`, thereby producing a cycle. Figure 9.9(b) shows a deceptively similar class diagram for a type that is possibly, but not necessarily, recursive. Given an instance of `C6`, it may contain a reference to an instance of `C7` or of `C8`, in keeping with the rule of subtype polymorphism. This holds as well for `C8`, `C9`, and `C10`. If the first reference was to an instance of `C8` and the second to `C10`, then there is a reference to `C6`, which would create a cycle. However, we cannot determine through analysis of the class diagram whether or not these conditions will be satisfied. Additionally, we observe that the implementation of this model as a recursive structure would require coercion, which is generally unsafe. We therefore do not consider the classes `C6`–`C10` to form a recursive type. This example illustrates a fundamental complication of mutually recursive types: there are cases where it is impossible to determine through class analysis whether the object structure resulting from a class diagram will contain recursive elements or not.

9.3 Clustering Techniques

The previous section explored the properties of class diagrams and their implications for the object diagram. In this section, we use these properties to show how object diagram patterns can be predicted through class diagram analysis.

Object Graph Cycles

Now that we have explored binary class relationships, multiplicity, and recursive types, we can integrate them to make some observations and assertions about the relationship between class diagrams and object diagrams. Specifically, we investigate cycles in the object graph and simple structures called *leaf clusters*. We present some interesting properties of these structures, along with outlines of their respective proofs.

An object graph is a directed graph, where the direction of an edge reflects which object is referencing which. A cycle in an object graph indicates a codependency among objects, a system of objects that require structural connectivity in order to interoperate. A cycle in an object graph can only be formed by aggregations in our formalism since only aggregations are realized as object attributes. Cyclic (non-aggregation) associations reflect an interdependency of functionality, but not necessarily a structural dependency.

Whether a graph has cycles or not is an important factor when choosing an algorithm to draw it. Most graph drawing techniques assume a directed acyclic graph [25], and so cycles must be removed before these techniques are applied [24]. However, through analysis of the class diagram, we can determine whether a program can produce cycles or not. Specifically, we can prove the following property of class diagrams and object diagrams (assuming no type coercion):

Property 9.10: Diagram Cyclicity. A cycle can occur in the object diagram if and only if the class diagram is recursive.

The proof of this property depends on the fact that edges in the object diagram are object references, and object references are the manifestation of aggregations in the class diagram. Furthermore, such a structural link between objects cannot occur unless there is a corresponding aggregation in the class diagram. The proof follows directly from these observations: for a cycle to exist in the object diagram, it must have corresponding cyclic aggregations; and if a recursive type is present in the class diagram, it can manifest as a cyclic object structure. As noted previously, it is not necessary that a recursive type always yield cycles.

Leaf Clusters

The distribution of objects and structural links in object-oriented systems obeys a power law: there are very few objects with high degree, and there are very many objects with low degree [73]. The empirical evidence supports the intuition that, in an object-oriented system, a few objects act as mediators and coordinators while many more objects act as simple data containers. We wish to define more formally a methodology for detecting these simple data objects so that they can be rendered appropriately in object diagrams. Specifically, since these data objects are tightly coupled with the objects that reference them, we wish to draw leaf objects in close proximity with their dependents.

Let $type(v)$ be a predicate that returns the most specific runtime type of an object v. Then we can define a leaf class as follows:

Definition 9.9. A **leaf class** in a class diagram is a class whose attributes, explicit and implied, are either primitive types or boxed types. We define a predicate $leafclass(C)$ that is true if C is a leaf class.

Our definition of a *leaf class* describes those classes of a class diagram that have no outgoing associations. Implied attributes are those that are not listed in the attributes portion of the class but exist nonetheless, and they include attributes that are shown as labeled aggregations and those that are inherited from superclasses. *Boxed types* are those immutable classes that provide object wrappers around primitive value types. The property of being a leaf class is not inherited, since subclasses of leaf classes can define attributes that invalidate the requirements of being a leaf class; however, a class' superclasses must be leaf classes in order for it to be one. In terms of the object graph, we can assert that the following property holds:

Property 9.11: Leaf Degree. The objects of a leaf class will be leaves in the object graph.

The proof of this property follows directly from our definitions. A leaf class has no outward aggregations by definition, including inherited ones. If an instance of a leaf class had an outgoing edge, then its class would have to have an outgoing aggregation, but this is a contradiction.

Definition 9.10. Given an object graph $G = (V, E)$, a **leaf cluster** is a subgraph $C = (V' \subseteq V, E' \subseteq E)$ such that the following four properties hold:

1. $V' = \{v*, v_1, v_2, \ldots, v_n\}$, where $n > 0$.
2. $leafclass(type(v_i))$ for $1 \le i \le n$.
3. $E' = \{(v*, v_i) \mid 1 \le i \le n \wedge (v*, v_i) \in E\}$.
4. $degree(v_i) = 1$ for all v_i where $1 \le i \le n$.

We call $v*$ the *leaf aggregator* and all other $v_i \in V'$ *leaf objects*. According to the definition, a leaf cluster is a two-level rooted tree in an object graph where all of the leaves are instances of a leaf class. Figure 9.1(b,d-e) shows a classic example of leaf clusters, where the `Data` class is the leaf class and `BST` is a leaf aggregator. The presence of compositions simplifies the detection of leaf clusters. Indeed, the very idea of a leaf cluster is closely related to the definition of composition: that an complex aggregate is made up simpler parts.

According to our definition, not all two-level rooted trees in the object graph are leaf clusters. This is necessary since program execution is dynamic, and the edges and vertices of the object graph can change with every execution step. An object that will eventually reference hundreds of other objects must at some point be created and initialized. At this point, we do not want to treat the object as if it were a leaf object, since we know from analysis of the class diagram that this is not the proper semantics of the object.

Let V be a set of classes and E be a set of directed relationships such that $G = (V, E)$ represents the structure of a class diagram. Then we can prove the following property of leaf clusters:

Property 9.12: Leaf Prediction. Leaf cluster classes can be identified in $O(|V| + |E|)$ time.

In order to predict which classes may be come leaf clusters, we first build a partial order of the vertices based on reversed generalizations. That is, we visit the most general class first and the most specific class last. This topological sorting can be done via a breadth-first search in $O(|V| + |E|)$ time. The set of classes can be divided into two distinct subsets: the set of leaf classes L and the set of non-leaf classes $\bar{L}$. This can be done in a single pass through the set of classes, as long as they are processed in order of decreasing generality. Each class and relationship will be analyzed at most once, checking if superclasses are in L or $\bar{L}$, giving time complexity of $O(|V| + |E|)$. Finally, those classes that aggregate leaf classes can be identified by searching backwards from part to whole along aggregations; this takes $O(|V| + |E|)$ time since in the worst case, all the classes may be leaf classes, and all aggregations would have to be followed.

Recursive Clusters

We showed previously that recursive types engender certain types of object structures. Within the context of that discussion, we looked at isolated recursive clusters; we did not consider the general case of structures that aggregate other clusters as well. In the worst case, a class diagram contains many aggregations and the resulting object diagram is a general graph. However, we can identify sets of classes that form *recursive clusters*.

Definition 9.11. Given an object graph $G = (V, E)$, a **recursive cluster** is a subgraph $C = (V' \subseteq V, E' \subseteq E)$ such that the following four properties hold:

- C is connected.
- for all vertices in V', either their types are participants in a single recursive type τ or they are leaf classes.
- for all edges $(u, v) \in E'$, either $type(v) \in \tau$ or
  $leafclass(type(v))$.
- there is at most one vertex $v \in V'$ such that there exist edges $(u, v) \in E$ where $u \notin V'$.

A recursive cluster is a subgraph of the object graph that is isolated, connected, and cohesive. It comprises a recursive object structure and the leaf objects aggregated by it. Furthermore, it has at most one entry point through which it is connected to the rest of the object graph. We make no other assertions about the structure of the recursive cluster. For example, we cannot assert whether recursive clusters are cyclic or not, although we can apply the technique from Section 9.3 to determine if it can possibly have cycles or not.

The key to predicting recursive clusters in the object diagram is the identification of recursive types in the class diagram. Let V be a set of classes and E be a set of directed relationships such that $G = (V, E)$ represents the structure of a class diagram. Then we can prove the following property of recursive types:

Property 9.13: Recursive Type Prediction. The classes in a recursive type can be identified in $O(|V \times E|)$ time.

Since recursive types are defined by generalized aggregation paths, proving this property requires us to have an algorithm for detecting generalized aggregation paths. Given a class diagram, a starting class c, and an ending class d, there is a generalized aggregation path between c and d if there is a path of aggregations and generalizations as given in Definition 9.7. This can be determined by a standard graph searching algorithm such as breadth-first. The execution of this search is constrained by the number of relationships, giving a running time of $O(|E|)$. A recursive type is a type that has a generalized aggregation path to itself; applying the technique just presented, this takes $O(|E|)$ time. Testing all classes in the class diagram therefore takes $O(|V \times E|)$ time. In practice, it is useful for this algorithm to return a set of recursive types, each of which is expressed as a list of class identifiers.

9.4 Clustering Algorithms

We divide the drawing algorithm into four stages:

1. *Class Diagram Processing:* The input is processed in order to extract information about simply recursive types and leaf classes.

2. *Object Graph Processing:* The current object graph is analyzed for structural properties.

3. *Clustering:* Each object is assigned to a cluster, and each cluster is drawn using local constraints.

4. *Arrangement:* The clusters are composed into a single drawing.

Class Diagram Processing

Class diagram processing is used to find leaf classes and 1-recursive types. This can be done using algorithms FindLeafClasses and FindSimpleRecursion, which are listed as Algorithms 9.1 and 9.2, respectively. The following utility functions are used by these algorithms:

- `TypeDepthSort(D)` takes a class diagram D and sorts the nodes in order of increasing type depth. This topological sorting is a partial ordering, since many types can have the same type depth.

- `asTree(D)` takes a class diagram D and returns the set of its classes. Given a class diagram with n classes, this function takes $O(n)$ time.

- `isLeafCandidate(c)` tests if a class c is a candidate for being a leaf class and returns a boolean. This is computed by testing whether c contains any fields that are not primitive, not `String`, and not of a boxed type. Given a class that contains f fields, this function takes $O(f)$ time.

- `removeTree(D,t)` removes a type t and all of its subtypes from a class diagram D. The worst case for this function occurs when removing a type that generalizes all other types in the hierarchy; therefore, given a class diagram with n classes, this function takes $O(n)$ time.

- `supertypeIn(t,S)` checks if any immediate supertype t' of t is in the set S. Given a type t with g supertypes, this function takes $O(g)$ time.

```
Input   : Class Diagram D
Output: Set of Leaf Classes
TypeDepthSort(D);
foreach class c ∈ D do
    if ¬isLeafCandidate(c) then
        removeTree(D,c);
    end
end
return asSet(D)
```

Algorithm 9.1: FindLeafClasses

```
Input   : Class Diagram D
Output: Set of 1-recursive classes
TopologicalSort(D);
S ← ∅;
foreach type t ∈ D do
    if supertypeIn(t,S) OR hasSelfLoop(t) then
        S ← S ∪ {t};
    end
end
return S
```

Algorithm 9.2: FindSimpleRecursion

Let n be the number of classes in a class diagram, D. Let m be the maximum number of fields in any class of D, and let g be the number of generalizations in D. Algorithm 9.1, *FindLeafClasses*, takes $O(n+m)$ time on D. Algorithm 9.2, *FindSimpleRecursion*, takes $O(n+g+m)$. Both functions are linear in time with respect to the size of the input.

Object Graph Processing

Leaf objects in the object graph are easily identifiable by finding the leaves of the graph (that is, those nodes with outdegree of zero) and determining if they are instances of a leaf class. We wish to couple leaf objects closely with their aggregate objects. To this end, we introduce a *modified object graph*, where nodes can be vertices or leaf clusters.

Definition 9.12. A **leaf cluster** is a set of objects composed of a nonempty set of leaf objects and the aggregator object. The type of a leaf cluster is considered to be the type of the aggregator. Formally, a leaf cluster is a tuple (v, L, E), where:

- v is the leaf aggregator object;
- L is a nonempty set of vertices, the *leaf objects*;
- E is nonempty set of directed edges (v_1, v_2) where $v_i \in \{v\} \cup L$ and $v_1 \neq v_2$.

Definition 9.13. A **modified object graph** is an object graph $G = (V, E)$, where V is a set containing vertices and leaf clusters and E is a set of directed edges. Edges that are contained entirely within a leaf cluster are not included in E.

The modified object graph is a specific type of object graph, and it can be treated as an object graph for the purposes of the other algorithms we present. Performing the transformation requires the definition of some utility functions and algorithms. We will use the function `type(v)` to return the runtime type of a vertex v.

Input : Object graph $G = (V, E)$,
Vertex $v \in V$,
Set of leaf classes L
Output: A boolean indicating if v is a leaf object or not

if $\mathit{type}(v) \in L$ **then**
  if $(\mathit{indegree}(v) = 0)$ **then**
    return *false*
  else
    let $u*$ be a vertex such that $(u*, v) \in E$;
    foreach $(u, v) \in E$ **do**
      if $u* \neq u$ **then**
        return *false*
      end
    end
  end
end
return *true*

Algorithm 9.3: IsLeafObject

Algorithm 9.3 determines if a vertex v represents a leaf object in an object graph. An object is a leaf object if it is an instance of a leaf class and if all incoming edges are from the same source, called $u*$ in the algorithm. In the worst case, all the edges of E will reference v and will need to be tested by the algorithm; the running time is therefore $O(|E|)$.

Algorithm 9.4 checks if a vertex is an aggregator of a leaf cluster. An object is a leaf aggregator if some of its adjacent vertices are leaf objects. It not required that all of a leaf aggregator's adjacent vertices be leaf objects. (along outgoing edges) are leaf objects. As with *IsLeafObject* (Algorithm 9.3), the worst case arises when we have to test all of the edges in E, and so this algorithm is also $O(|E|)$.

Algorithm 9.5 is used to create a modified object graph from a normal object graph. Each of the leaf aggregators are identified, and their respective leaf clusters are added to the modified object graph. Vertices that represent neither leaf objects nor leaf aggregators are added to the modified object graph without alteration. Line 15 serves to prevent redundant processing of vertices that

Input : Object graph $G = (V, E)$,
Vertex $v \in V$,
Set of leaf classes L
Output: A boolean indicating if v is a leaf cluster aggregator or not

if $outdegree(v) = 0$ **then**
    return *false*
else
    foreach $(v, u) \in E$ **do**
        if $IsLeafObject(G, u, L)$ **then**
            return *true*
        end
    end
    return *false*
end

Algorithm 9.4: IsLeafAggregator

Input : Object graph $G = (V, E)$,
Set of leaf classes L
Output: Modified object graph

$V' \leftarrow \emptyset$;
$E' \leftarrow E$;
$L_a \leftarrow \emptyset$;
foreach $v \in V$ **do**
    if $IsLeafAggregator(G, v, L)$ **then**
        $L_a \leftarrow L_a \cup \{v\}$;
        $E_c \leftarrow \emptyset$;
        $V_c \leftarrow \emptyset$;
        foreach $(v, v') \in E'$ **do**
            $E' \leftarrow E' - \{(v, v')\}$;
            $E_c \leftarrow E_c \cup \{(v, v')\}$;
            $V_c \leftarrow V_c \cup \{v'\}$;
        end
        $V' \leftarrow V' \cup \{(v, V_c, E_c)\}$;
        $V \leftarrow V - (\{v\} \cup V_c)$;
    else
        $V' \leftarrow V' \cup \{v\}$;
    end
end
return (V', E')

Algorithm 9.5: FormLeafClusters

were already considered.

In *FormLeafClusters*, we apply *IsLeafAggregator* (Algorithm 9.4) on each vertex in the original object graph. Even though the nested `foreach` loop of lines 9–13 operates over E, it will only process each edge once during the execution of the algorithm; this is because the edge selection is based on the source vertex's being the current vertex (v) under consideration, as selected by line 4. The running time of *FormLeafClusters* is therefore $O(|V||E|)$, roughly quadratic in the size of the input.

Clustering

A *cluster* is a subset of vertices in a graph. The vertices in a cluster will be drawn within distinct areas of the object diagram. We define three types of clusters in our approach, described below in order of decreasing specificity:

- *leaf clusters* are made up of leaf objects and their branch object.
- *simple structural clusters* are built from simple recursive object structures, where each object in the cluster is of the same type, modulo leaf objects embedded in leaf clusters.
- *graph clusters* are generic clusters that are not leaf clusters or homogeneous clusters.

The leaf clusters have already been formed by applying *FormLeafClusters* (Algorithm 9.5), as described in Section 9.4. What remains then is to separate the simple structural clusters from the rest of the objects. Algorithm 9.7, *Cluster*, presents our algorithm for identifying the simple structural clusters of a modified object graph. The algorithm returns a partitioning of the set of vertices into clusters; only the simple structural clusters are returned, and the non-clustered vertices can be determined by comparing the clustered nodes to those in the original graph. The algorithm itself requires the definition of some utility functions:

- `mark(v)` marks a vertex v as having been visited.
- `marked(v)` checks if a vertex v has been marked.
- `TopologicalSort(G)` orders the vertices of G according to a topological numbering. This can be done in lienar time.

Algorithm 9.6, *ConnectByType*, finds all of the vertices in an object graph that can be reached from a starting point. Furthermore, all of these reachable nodes must be of the same type as the starting point. In the worst case, all of the edges will have to be followed, and the result of

Input : Object graph $G = (V, E)$,
Vertex $v \in V$
Output: Set of vertices reachable from v where all have the same type

```
if marked(v) then
    return ∅
else
    mark(v);
    S ← {v};
    foreach (v, u) ∈ E do
        if type(u) = type(v) then
            S ← S ∪ ConnectByType(G, u);
        end
    end
    return S
end
```

Algorithm 9.6: ConnectByType

ConnectByType will be the set of all edges reachable from the starting point (regardless of type). The running time of the algorithm is therefore $O(|E|)$. *ConnectByType* works seamlessly with modified object graphs since we have previously defined the type of a leaf cluster as being the type of the cluster's aggregator; this captures cases such as that presented as the motivating example at the beginning of this chapter.

Input : Object graph $G = (V, E)$,
Set of simple recursive types R
Output: Set of simple structural clusters (a set of sets)

```
1  S ← ∅;
2  TopologicalSort(G);
3  foreach v ∈ V do
4      if v ∈ R then
5          T ← ConnectByType(G, v);
6          V ← V − T;
7          if |T| > 1 then
8              S ← S ∪ {T};
9          end
10     end
11 end
12 return S
```

Algorithm 9.7: Cluster

Algorithm 9.7 presents our technique for finding the simple structural clusters of an object graph. It starts with a topological ordering, and then all of the vertices are considered in turn. Once a vertex is considered, it is removed from V in order to prevent redundant processing. The set difference of line 6 ensures that nodes visited by *ConnectByType* are not also iterated by the `foreach` loop of lines 3–11. The running time of the algorithm is therefore $O(|V||E|)$, which like *FormLeafClusters* is roughly quadratic in the size of the input.

Arrangement in the Object Diagram

With the clusters determined, we can commence with the drawing of the graph. The simple leaf clusters will contain a relatively small number of vertices, and they can be drawn using highly customized algorithms. Given a leaf cluster that contains only two nodes – one aggregator and one leaf object – the two can be drawn tightly coupled, as shown in Figure 9.1. If there are multiple leaf objects, these can drawn using a simple hub or two-level tree layout. When simple leaf clusters are embedded within simple structural clusters, the cluster itself can be treated as a node defined by the bounding box of the cluster.

There is an important distinction to be made between our clustering approach and the clustered drawings of systems such as `dot`[33]. `dot` requires the definition of clusters on input, and it draws each cluster using the same algorithm. We do not restrict the drawing of clusters in such a way. If it is known that an alternative drawing strategy is superior for a specific cluster's substructure, then such a strategy can be applied. For example, if it is known (due to user input or inference) that a cluster forms a tree, or a list, or a grid, then appropriate drawing algorithms can be applied within the cluster. The cluster itself then is treated as a single large vertex within its enclosing drawing, which itself may be a cluster in its enclosing drawing, and so on.

The simple structural clusters can similarly be analyzed in order to determine how they should be drawn. For example, if the contents of the cluster forms a linear succession of nodes (as with a linked list implementation), then the cluster should be drawn as a list. In the default case, simple layered graph drawing algorithms can be used, since they produce aesthetically decent drawings in general, even though they do not necessarily represent the relationships present in the original source code or class diagram [85, 24]. Developing criteria to decide which algorithms to apply to which structures is an area of continued research.

The *top-level clusters* are any cluster besides leaf clusters that are nested within another cluster. In order to arrange the top-level clusters in the diagram, each cluster can be treated as a graph node whose size is determined by the cluster's bounding box. As with individual cluster drawing, we are investigating means by which the best algorithm can be chosen for laying out clusters based on the class diagram. Our current approach is to use orthogonal drawing techniques to organize the top-level clusters.

Arrangement in the Sequence Diagram

This clustering technique can be used in sequence diagrams as well as object diagrams. In Chapter 8, we presented two techniques for arranging lifelines in the sequence diagram: the simulated annealing approach and the greedy approach. Each approach has its own set of benefits and drawbacks, and neither produces optimal results with respect to aesthetic preferences and computational efficiency. The JIVE methodology uses the first-come first-served technique due to its speed and simplicity. Figure 8.7 illustrated the major problem of the greedy approach, that it entails no prevention of undesirable back edges. The program used to generate the screenshot of Figure 8.7 is an application that generates binary search trees whose nodes each reference a `StringElem` object. In this program, and as shown in the sequence of the screenshot, the `StringElem` data elements are created before their corresponding `Tree` node objects; hence, each time a the `insert` method is called on any `Tree`, it will delegate to its `StringElem` and a back edge results. However, we notice here that `Tree` and `StringElem` objects form leaf clusters, where the `Tree` is the aggregator class and `StringElem` is the leaf class. In Figure 8.7, each `StringElem:iTree:i` pair forms a leaf cluster.

The major disadvantage of the simulated annealing approach in interactive execution is that the process must be repeated when the data model changes. For example, when a new lifeline is added, the annealing algorithm must be executed in order to determine where the lifeline should be placed. However, there is no guarantee that the other lifelines will be in their same locations, and since many lifelines may have their positions shifted by the insertion of the new lifeline, there is substantial danger of the user losing his or her mental map of the diagram. However, with the introduction of clusters, we can allow disciplined transposition of lifeline positions, minimizing the the potential of mental map loss while promoting the aesthetic preference against back edges.

The cluster-aware technique for drawing the sequence diagram is called the *greedy technique* since it seeks local optimizations within clusters. Using this technique, we allow lifelines to be moved within their clusters, and in this way, related lifelines can be kept together while allowing for more sophisticated handling of other aesthetic preferences. Given a cluster that contains n objects, we can use the simulated annealing algorithm within the cluster in order to determine the optimal arrangement of lifelines. Lifelines whose objects are not part of leaf clusters can be treated as belonging to singleton clusters. Then there are two options: use a hybrid greedy-annealing approach or use a cluster-aware annealing appreach. In the hybrid approach, the greedy technique is used until the number of back edges exceeds a user-specified threshhold, at which time the simulated annealing algorithm is applied on a subset of the object diagram: it is only applied within leaf clusters individually. Since these clusters should be quite small, simulated annealing's relative inefficiency becomes less of a concern. In the cluster-aware annealing approach, we augment Algorithm 8.2 to operate using clusters. Specifically, when lifelines are randomly chosen by the algorithm, it is illegal to move them outside of their clusters, and furthermore, clusters can be moved as units. In this way, simulated annealing can be used without breaking the constraint that leaf clustered objects are adjacent in the sequence diagram; however, this non-hybrid approach can potentially cause dramatic changes in the diagram if there are relatively few leaf clusters. The modified algorithm is presented here as Algorithm 9.8.

Input: Sequence graph G, Objective function F, Initial temperature t, Cooling factor α, Minimum temperature m
Output: Labels for each object lifeline

repeat
    $u \leftarrow$ random vertex of G;
    $v \leftarrow$ random vertex of G;
    if *u and v are in the same cluster* **then**
        compute F assuming labels of u and v are swapped
    else
        compute F assuming we swap the labels of the cluster containing u and the cluster containing v;
    end
    $\delta \leftarrow$ (value of F with exchange) $-$ (current value of F);
    With probability $\min(1, exp(-\delta/t))$, swap the labels according to whether u and v are in the same cluster;
    $t \leftarrow \alpha t$;
until $t \leq m$;

Algorithm 9.8: ClusteredAnnealing: A cluster-preserving modification of the simulated annealing approach to arranging object lifelines in a sequence diagram. This is a modification of Algorithm 8.2.

9.5 Extended Clustered Object Diagram Example

Once the class diagram and object graph are analyzed, the object diagram must be constructed. We differentiate here between the *object graph*, which is a mathematical abstraction, and the *object diagram*, which is a drawing of the object graph. There can be more than one object diagram for any object graph. The methodology we present in this section outlines the approach we have taken.

The overall strategy for layout involves the formation of *clusters* of vertices and the arrangement of these clusters [81]. We use the two types of clusters defined previously: leaf clusters and recursive clusters. The clusters themselves can then be treated as vertices with non-trivial size, and they can be arranged using existing techniques.

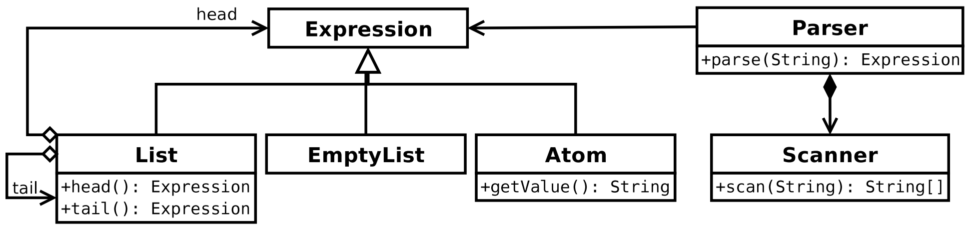


Figure 9.10: An architecture for a simple Lisp-style expression parser.

In this section, we demonstrate how our approach works through an example. Figure 9.10 presents a UML class diagram for a simple expression parser. The expressions are list-based in the Lisp tradition, where each expression is a `List`, `EmptyList`, or `Atom`. `List` objects have a `head` and a `tail`, both of which are themselves expressions. The `Parser` takes a string as input and parses it into an expression with the help of a `Scanner`.

Cluster Formation

The static analysis of the class diagram requires two steps: identifying the leaf classes and finding recursive types. In our example, there are three leaf classes: `Scanner`, `EmptyList`, and `Atom`. Each of these classes has no outgoing aggregations, nor do they inherit any from their superclasses. There is one recursive type in Figure 9.10, the set {`Expression`, `List`}. `List` aggregates itself, making it a simple recursive type.

Figure 9.11 shows an object graph representing a state in the parser. This drawing was generated using a top-down layering followed by barycenter crossing-reduction and horizontal coordinate assignment step [24, 85]. At the visualized state of the program, the parser has built an expression tree for input of the form `((A B) C D)`, where `A`, `B`, `C`, and `D` are atoms. As predicted, the graph verifies that the instances of leaf classes are leaves in the graph. A recursive object structure has indeed arisen from `List` and `Expression`. We are not drawing all of the objects on the heap, only those objects whose classes are shown in the class diagram. In a language such as Java, it is nigh-impossible to draw the complete heap since even the simplest program has thousands of objects in memory at runtime.

Cluster Arrangement

Figure 9.12 shows a final drawing of the object graph where the clusters are laid out using an orthogonal drawing technique. Each leaf cluster is drawn using the same orthogonal technique, and the simple structural cluster is drawn using layered drawing techniques, as in the original drawing of

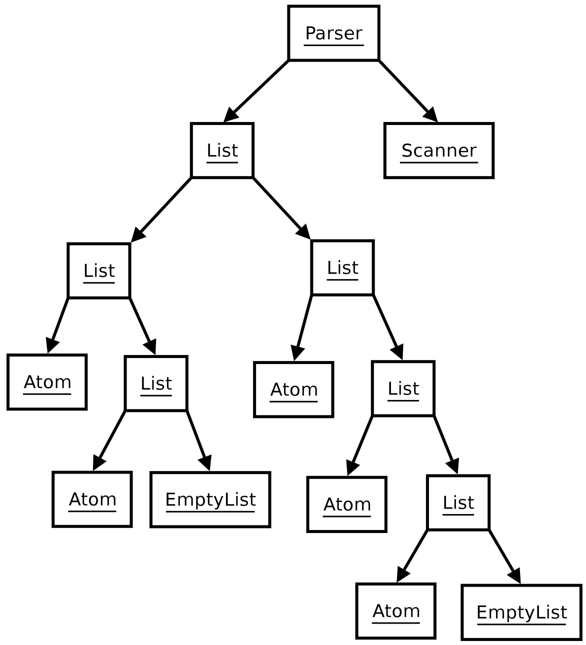


Figure 9.11: An object diagram for the Parser example. This drawing was produced using standard layered graph drawing techniques.

Figure 9.11. The clusters have been aligned to enforce a straight-line drawing. In an implementation, a more general approach involving edge routing between clusters may be required.

Figure 9.13 provides an alternate view of the drawing where the leaf clusters are shown in grey boxes and the simple structural cluster is shown in a dotted rectangle. This view highlights how the different drawing techniques are used in each cluster and how these combine to form a good drawing.

It is important to notice that the drawings of Figures 9.12 and 9.13, are not optimal with respect to overall drawing size. The standard layered drawing technique shown in Figure 9.11 produces drawings that are compact in area. Our technique weakens this constraint in order to achieve more visible distinction between components. In Figure 9.12, we can see a clear separation between the expression tree, which is a simple recursive type, and the parser/scanner component.

The essence of our technique is in the declarative formation of clusters in the object graph. In practice, these clusters can be drawn many different ways. The leaf clusters of Figure 9.12 are drawn using a simple algorithm where the leaf aggregator is drawn on the left and leaf objects are drawn to the right of the aggregator. This technique is sufficient for our example, but a more robust solution would involve more advanced graph drawing techniques such as radial drawing [24], 2.5-dimensional drawings, or focus-directed hierarchical drawing [16].

Before drawing recursive clusters, we first remove its leaf clusters and replace them with vertices

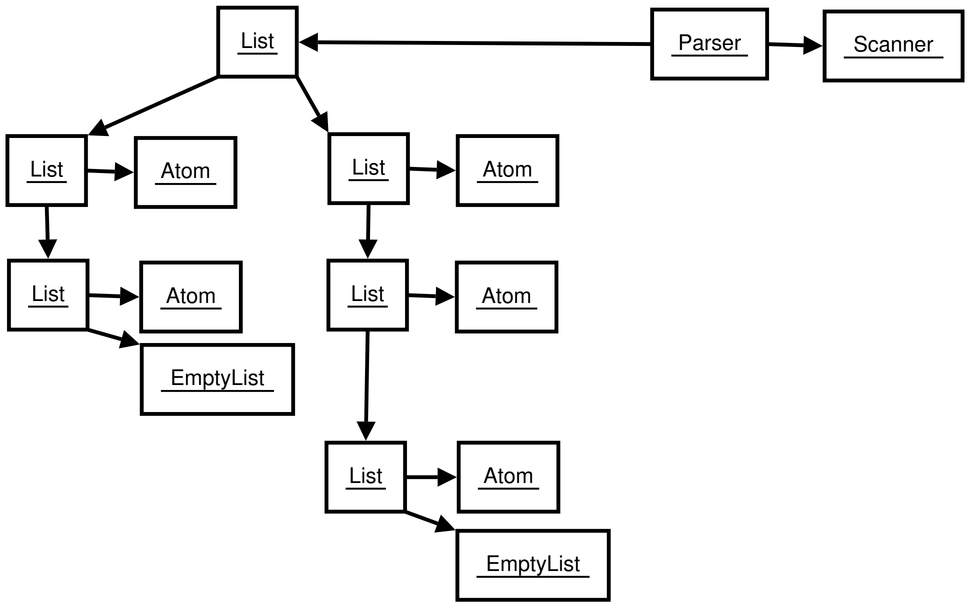


Figure 9.12: A drawing of the expression parser object graph using our techniques of class diagram and graph analysis

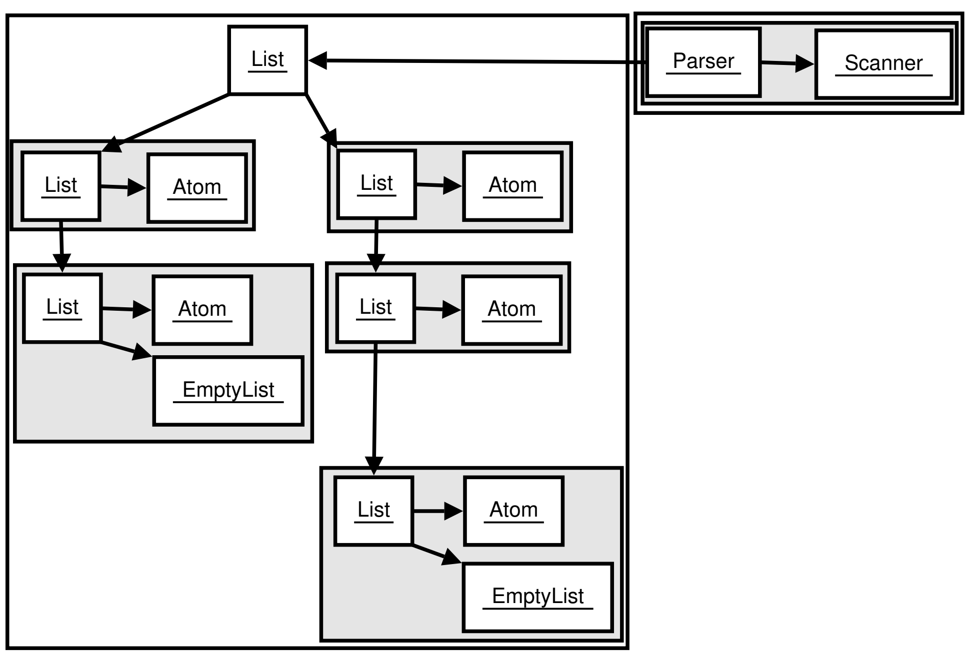


Figure 9.13: An alternate view of the drawing in Figure 9.12. In this view, leaf clusters are drawn in grey boxes and simple structural clusters are drawn in dotted rectangles.

of nontrivial area. This technique follows from Definition 9.11, in which we allow leaf clusters to be contained with recursive clusters. We have drawn the recursive cluster using a hierarchical technique [85]. This has produced a good drawing since the recursive cluster is itself a tree. In general, this drawing step would require converting the graph into a directed acyclic graph first. Other techniques can be used to arrange clusters as well, including blob layouts [46] or other orthogonal methods [24].

Chapter 10

Conclusions and Future Work

This dissertation represents a significant contribution to the study of object-oriented programming and interactive program visualization. The major contributions of this work are: a visual operational semantics for Java, created using a modification and modernization of the contour model; an architecture for interactive visualization and forward- and reverse-execution of Java programs; the application of graph-drawing research towards novel approaches for drawing enhanced object and sequence diagrams, including a combination of graph-theoretic techniques with program-specific properties. In addition to the theoretical contributions, the JIVE tool is an artifact of this research, a realization of many of the ideas presented in this work. This research has proven to be fertile ground for new ideas, revealing new directions and possible projects.

In Chapter 1, we presented seven desiderata for the visualization of object-oriented Java programs. In this chapter, we revisit this list and demonstrate how this dissertation successfully addresses each desideratum. A summary of future work, both long- and short-term, is also provided.

10.1 Desiderata Revisited

1. Depict Objects as Environments. The visual operational semantics presented in Chapter 3 represents a significant contribution to program visualization. This visual language builds upon the fundamental work of Jayaraman and Baltus [53], which introduced the formal visualization of objects as environments of execution. The notation presented herein substantially extends the previous work to provide coverage of the Java programming language. In addition to the essential object-oriented constructs of object and method, our notation covers Java's unique separation of static and instance space, its inner objects (named, anonymous, and static), and its integrated support for multiple threads.

2. Provide Multiple Views of Execution State. Our taxonomy of views was presented in Chapter 3, and it includes detailed view, minimized view, compact view, and call-path view. JIVE provides the user with the freedom to use a single view or to combine views in any desired fashion; for example, a portion of the diagram can be minimized while the rest is viewed in detail.

3. Capture History of Execution and Method Interaction. With JIVE, we introduce dynamically-generated sequence diagrams that represent actual program execution history. This automatic generation of sequence diagrams and the corresponding aesthetic analysis is a significant contribution. JIVE is among the first systems to include automatically-generated sequence diagrams (see Malloy and Power [67] for a recent work with dynamically visualization of C++); no other studies have been published on the aesthetics of automatic sequence diagram drawing.

4. Support Forward and Backward Execution. The dynamic contour model provides a formal model for interactive execution, and the JIVE provides an effective implementation. JIVE employs incremental state saving, which provides the most efficient model for incremental steps between consecutive states.

5. Support Queries on the Runtime State. The JiveLog module provides a database abstraction over a program's execution history. This database layer can be queried using standard query languages such as SQL, and these queries can be posed through a graphical interface. This fifth desideratum is the only one for which the progress towards it is still preliminary. This is further described in Section 10.2 and by Girgis et al. [39]. It is important to note that although the graphical query interface is preliminary, the technological underpinnings are already established through the JiveLog database.

6. Produce Clear and Legible Drawings. The techniques presented in Chapters 7 and 8 provide formal, primarily graph-theoretic approaches to drawing enhanced object diagrams and sequence diagrams, respectively. While these are significant in their treatment of the respective aesthetics of the diagrams, further significant contribution is made in Chapter 9. We have shown that an analysis of a program's class diagram, specifically the identification of recursive and leaf classes, can yield significantly better drawings of both object and sequence diagrams without impacting the algorithms' asymptotic complexities.

7. Use Existing Java Technologies. The implementation of the JIVE methodology is based upon existing Java technologies; no custom compiler or virtual machine is necessary. JIVE deals with unmodified Java programs, and no special source-code annotations are required. This is made possible through effective use of the Java Platform Debugger Architecture (JPDA) and Apache's Byte Code Engineering Library (BCEL).

10.2 Current Status and Future Work

In this section, we describe the status of the JIVE tool, some domains in which it has been used, and some directions for future work. Some of the work described herein is currently being investigated by the JIVE research group at the University at Buffalo.

Architecture and Applications

We have used JIVE and its notations in numerous university courses, including both undergraduate and graduate courses. As a pedagogic tool, the visual operational semantics for Java have proven useful for explaining the concepts of object-oriented programming. There have been several graduate-level seminars dedicated to the exploration of interactive visualization of object-oriented programs. This has included a study of the dynamic, runtime behavior of design patterns as visualized through JIVE. Some of the examples produced from these studies have been presented in Chapter 4. Most of these studies have been focused on the development of JIVE, the JIVE methodology, and researching new directions for the tool. Since the feature set is still not stable and the tool is still under active development, there has not been any formal analysis of pedagogic or debugging impact. It is desirable to build upon our anecdotal experience with a more formal study.

Chapter 5 described the history of the JIVE architecture from processing a limited, custom programming language, to a pure Java source-code transformation technique, to the modern JPDA-based methodology. This mature architecture is a significant contribution, and it is proven to have significant scalability since it is built upon the same underlying technology as all modern Java debuggers. However, there are interesting possibilities for extending and modifying the architecture to provide new possibilities. Although it is technically against our desiderata for visualization, it is conceivable to engineer a related visualization tool that is built upon a custom virtual machine that incorporates extended functionality. For example, NASA has recently released its JavaPathFinder technology as open source[1]. JavaPathFinder is a custom virtual machine that verifies bytecode by providing an explicit state software model checker. Integrating JavaPathFinder and JIVE could produce a powerful debugging and program understanding tool. Similarly, it would be an interesting architectural challenge to integrate JIVE into an IDE such as Eclipse; such an endeavor would

[1] `http://javapathfinder.sourceforge.org`

certainly broaden JIVE's appeal and increase its rate of acceptance.

Graph Drawing

Chapter 7 presented our approach for drawing enhanced object diagrams. Our technique involves the conversion of the contour model object structure into a graph, generating a drawing of the graph, and then reversing the transformation to yield an object diagram. We identified the specific properties of the contour model that make this a challenging problem, such as heterogeneous contours and edges, meaningful crossings, and nested substructures.

Our experimentation with JIVE's drawing system have thus far focused on two types of drawings: grid-based drawings and hierarchical layered drawings. The grid-based drawings were used in a very early version of JIVE, using a layout approach similar to Java's `FlowLayout` class. In this approach, objects are laid out in a first-come, first-served fashion, filling horizontal column while there is space, then moving to the next vertical row when necessary. Despite the relative success of a similar first-come first-served approach in the layout of object lifelines in automatically-drawn sequence diagrams (see Section 8.4), this approach was not very effective in drawing meaningful object structures; this was a predictable result, however, since the technique essentially ignores graph drawing aesthetics. The layered drawing technique has proven much more useful, especially in the visualization of hierarchical and recursive structures such as trees. We showed that with a clever conversion from contour model to graph, existing hierarchical drawing techniques can be applied to produce aesthetically good drawings.

There are many other graph drawing paradigms that could be applicable to drawing JIVE's enhanced object diagrams. Orthogonal drawings and radial drawings can be used, and flow-based drawings may also be used. It is likely that different drawing techniques produce aesthetically variable diagrams, and that these aesthetic results are also dependent upon the type of data being visualized. Further exploration and experimentation is required on this front. First, additional drawing paradigms would have to be adapted to the contour model, and then these would have to be implemented within JIVE. This would enable a comprehensive study of multiple drawing paradigms and their applicability on different types of data.

We presented in Chapter 8 two techniques for arranging object lifelines. The simulated annealing approach is stochastic and its running time depends on user-specified input values. The first-come first-served approach has worked well in practice, although it has not been rigorously tested in a variety of programs; the results are mostly anecdotal. In the current version of the JIVE tool, only the greedy approach has been implemented. It is desirable to be able to switch dynamically between the two drawing techniques and to formally measure how the two approaches satisfy our sequence diagram aesthetics on a various types of programs.

Chapter 9 presented our work on combining program-specific properties with graph-theoretic properties. We defined recursive clusters and leaf clusters as object structures that are formed by specific class diagram patterns; this makes their presence at runtime predictable through static analysis. We showed how this clustering can be used to generate drawings that more closely reflect the semantics of the program being visualized. Recursive and leaf clusters were defined based on a study of design patterns and our own understanding of object-oriented program patterns and structures. It is likely that there are other such patterns that can be exploited to generate better drawings. Furthermore, it may be possible to detect design patterns at runtime and use specialized heuristics to draw the constituent objects in a manner that reflects the semantics of the pattern itself. For example, a recursive cluster is closely related to the *Composite* design pattern; other types of clusters may correspond to other patterns as well.

Our drawing efforts thus far have focused on static drawings, but JIVE is a dynamic environment. There are certainly applications of dynamic graph drawing within the JIVE methodology, but these have not yet been explored. The set of transactions can be interested as the possible changes to the data model, making them analogous to the graph modifications studied in dynamic graph drawing [see *e.g.* 22]. Experimentation in this direction would require a formalization of the relationship between dynamic contour models and dynamic graphs and then the implementation of these techniques within the JIVE framework.

Visual Queries and Data Mining

JIVE research has introduced the novel perspective of program execution as a database. This is realized in the JIVE architecture through JiveLog and JiveLogDB, as presented in Chapter 5. On one hand, the database perspective is an implementation convenience since databases provide robust scalability, allowing JIVE to handle larger datasets than would be possible with a strictly in-memory representation. However, the database perspective provides a much deeper and richer possibility: the capability for querying program execution through a database query language. JiveLogDB stores the complete contour-centric history of program execution, which is a rich collection of data for queries.

JIVE queries are not limited to text-based queries through a query language such as SQL. JIVE provides visualizations of program state and program history, and these visualizations can be used as visual interfaces into an integrated query system. Some of the capabilities of such a system have been described by Girgis et al. [39], but this work is still in a preliminary stage, though it is actively being researched. Some examples of queries and their graphical annotation are provided below. These queries can be combined with JIVE's interactive execution functionality to seamlessly move between program states by navigating through the sequence diagram.

- *What variables hold a specific value?* The answer of this query is a set of variables, which can be highlighted in the object diagram. Additionally, the times at which the variables hold the values can be annotated on the sequence diagram.

- *What are the values of a variable?* The answer to this query is a list of values, which can be shown in a table. Additionally, the sequence diagram can be annotated with the times at which the variable held different values.

- *What are the instances of a specific class?* The answer to this type of query is a list of objects, and the objects can be highlighted in the object diagram and the sequence diagram. Additionally, the sequence diagram can be annotated with the times that the objects were created.

- *When was a specific line of code executed?* This query be answered by annotation of the sequence diagram and by highlighting the corresponding method contours in the object diagram, as shown in Figure 10.1, an artist's conception of the graphical query interface.

- *What methods were executing when a variable was changed?* The result of this query is a set of methods for each variable change, or a set of sets of method activations. These can be annotated on the sequence and object diagrams as described above, but it is important to note that this is inherently a more complex query than the others: it uses two different kinds of queries — variable modification and method execution over time — to form a compound query.

This sampling of queries provides only a rough idea of the future of JIVE's graphical query system. Using the JiveLogDB format, any query that can be posed in SQL can be posed over the execution history; the research challenge for future work is to wrap these queries effectively and efficiently within the visualization framework of JIVE.

To an extent, a system that queries the runtime behavior of a system is similar to a post-mortem assertion checker. Assertions are used during program execution to ensure that invariants hold;

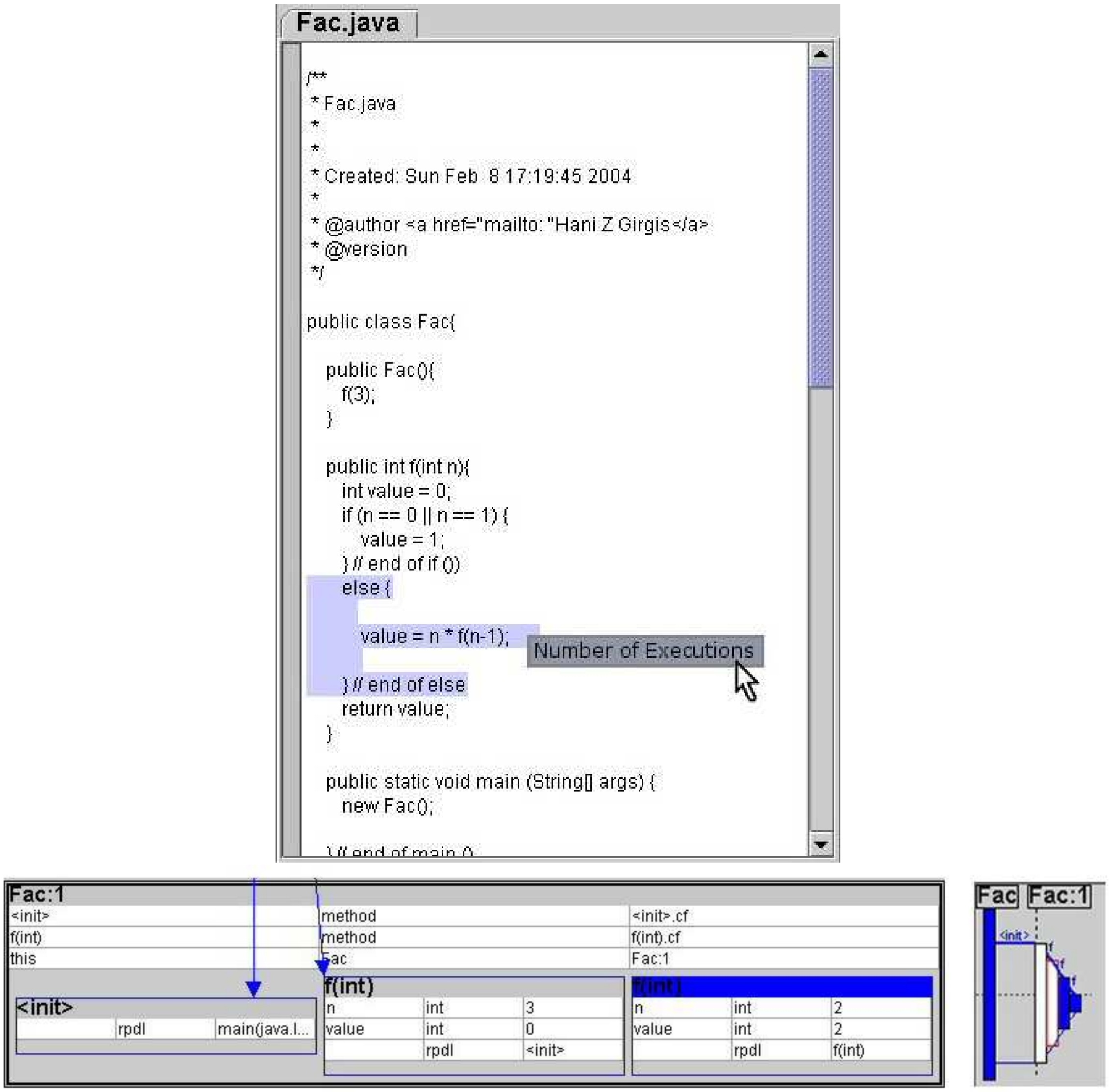


Figure 10.1: An example of a graphical query in JIVE regarding the number of executions of a specific line of code. This selected method computes the factorial of its input, and the query is testing when the "`n*f(n-1)`" case is executed. The answer is highlighted in the sequence diagram in the lower-right of the figure: the two method activations that ran the selected line are shown in white. Furthermore, the corresponding method contours are shown in the object diagram in the lower-left of the figure.

queries over program execution can be used in a similar way to determine where conditions are upheld and where they are broken. By formalizing the desired invariants within the query system, it becomes easier to locate possibly erroneous states.

The database perspective of program execution history also enables data mining over one or more executions of a program. Given a JiveLog trace, analyses can be run to collect standard software metrics, such as the number of times a method was called or the number of objects created of a monitored class. It should be noted that JIVE does not support collection of memory usage or time-related metrics; these would be inherently skewed by the JIVE methodology itself, which uses more memory and time than a program running without visualization. However, through JiveLog, more complex trace information can be collected of the sort discussed with queries above. For example, one could inquire what methods were running at the time specific variables were modified. This type of analysis is not possible with a standard debugger or software visualization system. Furthermore, queries and analyses can be run over multiple JiveLogs of the same or even different programs. This has important implications for understanding program execution, and it can lead to other domains such as security analysis: it is theoretically possible to build a knowledge base of "normal" program executions and then use runtime analysis to detect and isolate programs exhibiting abnormal activity.

Closing Remarks

JIVE has provided a fertile ground for interesting and significant research. While there has been a great deal of work done, there are still many open avenues for new direction. The modern JIVE tool started as a program for drawing the stack and heap structures of a language fabricated to explain the semantics of parameter passing. It evolved into a tool to visualize Java states, and then to an environment that couples state and history visualizations. We are now on the cusp of introducing a new paradigm for debugging, using visual queries and results within an environment for interactive execution. There are many possibilities for future work, including enhanced automatic graph drawing, incorporation of slicing and static analysis to predict structures and patterns, and optimizations in the interactive execution engine. This dissertation is built upon the work of many, and has been assembled with the help of many; it is my earnest desire that this work be a continuation, that others may build upon it and explore this fascinating research field of interactive visualization of object-oriented languages.

Bibliography

[1] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23(6):589–616, 1993. ISSN 0038-0644.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison–Wesley, 1983.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 311–330. ACM Press, 2002. ISBN 1-58113-471-1.

[4] E. Allen. *Bug Patterns in Java*. Apress, 2002.

[5] S. W. Ambler. *The Elements of UML Style*. Cambridge University Press, 2002.

[6] J. W. Atwood, M. M. Burnett, R. A. Walpole, E. M. Wilcox, and S. Yang. Steering programs via time travel. Technical Report 95-60-08, 1995.

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

[8] D. J. Barnes and M. Kölling. *Objects First with Java: A Practical Introduction using BlueJ*. Prentice Hall / Pearson Education, 2nd edition, 2004.

[9] A. Barr. *Find the Bug: A Book of Incorrect Programs*. Addison–Wesley Professional, 2004.

[10] D. B. Baskerville. Graphic presentation of data structures in the dbx debugger. Technical report, 1985.

[11] C. Batini, L. Furlani, and E. Nardelli. What is a good diagram? a pragmatic approach. In *Proceedings of the Fourth International Conference on Entity-Relationship Approach*, pages 312–319. IEEE Computer Society, 1985. ISBN 0-444-87951-X.

[12] C. Batini, E. Nardelli, and R. Tamassia. A layout algorithm for data flow diagrams. *IEEE Trans. Softw. Eng.*, 12(4):538–546, 1986. ISSN 0098-5589.

[13] M. R. Birch, C. M. Boroni, F. W. Goosey, S. D. Patton, D. K. Poole, C. M. Pratt, and R. J. Ross. Dynalab: a dynamic computer science laboratory infrastructure featuring program animation (abstract). In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pages 29–33. ACM Press, 1995. ISBN 0-89791-693-X.

[14] B. Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Not.*, 34(4):61–69, 1999. ISSN 0362-1340.

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

[16] F. Boutin and M. Hascoët. Focus dependent multi-level graph clustering. In *Proceedings of the working conference on Advanced visual interfaces*, pages 167–170. ACM Press, 2004. ISBN 1-58113-867-9.

[17] M. Brörkens and M. Möller. Dynamic event generation for runtime checking using the jdi. *Electronic Notes in Theoretical Computer Science*, 70(4):1–15, 2002.

[18] M. H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *IEEE Workshop on Visual Languages*, pages 4–9, October 1991.

[19] V. Cerny. A thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[20] Z. Chen and B. Xu. Slicing object-oriented java programs. *SIGPLAN Not.*, 36(4):33–40, 2001. ISSN 0362-1340.

[21] E. G. Coffman and R. L. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, 1(3):200–213, 1973.

[22] R. F. Cohen, G. Di Battista, R. Tamassia, I. G. Tollis, and P. Bertolazzi. A framework for dynamic graph drawing. In *SCG '92: Proceedings of the eighth annual symposium on Computational geometry*, pages 261–270. ACM Press, 1992. ISBN 0-89791-517-8.

[23] J. Cook. Reverse execution of java byte-code. *Computer Journal*, 45(6):608–619, 2002.

[24] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for The Visualization of Graphs*. Prentice Hall, Upper Saddle River, New Jersey, 07458, 1999.

[25] J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Comput. Surv.*, 34 (3):313–356, 2002. ISSN 0360-0300.

[26] P. Eades and D. Kelly. Heuristics for reducing crossings in 2-layered networks. *Ars Combin.*, 21.A:89–98, 1986.

[27] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics 91*, pages 24–33, 1991.

[28] P. Eades, T. Lin, and X. Lin. Two tree drawing conventions. *International Journal of Computational Geometry and Applications*, 3(2):133–153, 1993.

[29] P. Eades, X. Lin, and W. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 6:319–323, 1993.

[30] M. Eiglsperger, M. Kaufmann, and M. Siebenhaller. A topology-shape-metrics approach for the automatic layout of uml class diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 189–ff. ACM Press, 2003. ISBN 1-58113-642-0.

[31] U. Ferraro-Petrillo. *JIVE: Java Interactive software Visualization Environment User Manual*, June 2003. URL `http://jive.dia.unisa.it`.

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

[33] E. Gansner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[34] M. Garey and D. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco, 1979.

[35] M. R. Garey and D. Johnson. Crossing number is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 4:312–316, 1983.

[36] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM Journal on Computing*, 31(2):601–625, 2001.

[37] 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.

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

[39] H. Z. Girgis, A. V. Hegde, M. Pushpendran, P. V. Gestwicki, and B. Jayaraman. Visual queries for interactive execution of java programs. Technical Report 2004-19, Department of Computer Science and Engineering, University at Buffalo, December 2004.

[40] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN 0-201-11371-6.

[41] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[42] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: putting icing on the uml cake. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 301–314. ACM Press, 2004. ISBN 1-58113-831-9.

[43] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. A new approach for visualizing uml class diagrams. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 179–188. ACM Press, 2003. ISBN 1-58113-642-0.

[44] J. Hahn and J. Kim. Why are some diagrams easier to work with? effects of diagrammatic representation on the cognitive intergration process of systems analysis and design. *ACM Trans. Comput.-Hum. Interact.*, 6(3):181–213, 1999. ISSN 1073-0516.

[45] J. Hamer. Visualising java data structures as graphs. In *CRPIT '03: Proceedings of the sixth conference on Australian computing education*, pages 125–129. Australian Computer Society, Inc., 2004.

[46] D. Harel and G. Yashchin. An algorithm for blob hierarchy layout. In *Proceedings of the working conference on Advanced visual interfaces*, pages 29–40. ACM Press, 2000. ISBN 1-58113-252-2.

[47] W. Harrison, C. Barton, and M. Raghavachari. Mapping uml designs to java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 178–187. ACM Press, 2000. ISBN 1-58113-200-X.

[48] 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 35th SIGCSE technical symposium on Computer science education*, pages 387–391. ACM Press, 2004. ISBN 1-58113-798-2.

[49] A. Holub. *Holub on Patterns: Learning Design Patterns by Looking at Code*. APress, 2004.

[50] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of squeak, a practical smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326. ACM Press, 1997. ISBN 0-89791-908-4.

[51] T. Jacobs and B. Musial. Interactive visual debugging with uml. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 115–122. ACM Press, 2003. ISBN 1-58113-642-0.

[52] B. Jayaraman. Fundamentals of programming languages, 1999. Lecture notes, University at Buffalo Department of Computer Science and Engineering.

[53] B. Jayaraman and C. Baltus. Visualizing program execution. *Proc. IEEE Symp. on Visual Languages*, pages 30–37, September 1996.

[54] D. F. Jerding and J. T. Stasko. Using visualization to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Atlanta, GA, USA, July 1994.

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

[56] B. Joy, G. Steele, J. Gosling, and G. Bracha. *Java Language Specification*. Addison–Wesley, 2nd edition, 2000.

[57] M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1:1–25, 1997.

[58] A. Kay. The early history of smalltalk. *ACM SIGPLAN Notices*, (3):69–95, March 1993.

[59] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983. URL `citeseer.ist.psu.edu/kirkpatrick83optimization.html`.

[60] D. E. Knuth. The errors of tex. *Softw. Pract. Exper.*, 19(7):607–685, 1989. ISSN 0038-0644.

[61] R. Kollmann and M. Gogolla. Application of UML associations and their adornments in design recovery. In *Working Conference on Reverse Engineering*, pages 81–91, 2001.

[62] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison–Wesley, third edition, 2003.

[63] T. Lin. *A General Schema for Diagrammatic User Interfaces, Ph.D. Thesis*. The University of New Castle, New Castle, Australia, 1994.

[64] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison–Wesley Professional, 2nd edition, 1999.

[65] M. A. Linton. The evolution of dbx. *Proceedings of the Summer USENIX Conference*, pages 211–220, June 1990.

[66] J. Lönnberg, A. Korhonen, and L. Malmi. Mvt: a system for visual testing of software. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 385–388. ACM Press, 2004. ISBN 1-58113-867-9.

[67] B. A. Malloy and J. F. Power. Exploiting uml dynamic object modeling for the visualization of c++ programs. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 105–114, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-073-6. doi: http://doi.acm.org/10.1145/1056018.1056033.

[68] R. C. Martin. Uml tutorial: Sequence diagrams (engineering notebook column), 1998. Engineering Notebook Column from Object Mentor, Inc.

[69] S. Mukherjea and J. T. Stasko. Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger. *ACM Trans. Comput.-Hum. Interact.*, 1(3):215–244, 1994. ISSN 1073-0516.

[70] S. C. North. Incremental layout in dynadag. In *Graph Drawing*, pages 409–418, 1995. URL `citeseer.ist.psu.edu/north96incremental.html`.

[71] J. Petit. Experiments on the linear arrangement problem. *Journal of Experimental Algorithmics*, 8, 2003.

[72] A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. In *ASWEC '04: Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*, page 251. IEEE Computer Society, 2004. ISBN 0-7695-2089-8.

[73] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in object-oriented programs. *Communications of the ACM*, May 2005.

[74] H. C. Purchase, R. F. Cohen, and M. James. Validating graph drawing aesthetics. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 435–446. Springer-Verlag, 1996. ISBN 3-540-60723-4.

[75] H. C. Purchase, M. McGill, L. Colpoys, and D. Carrington. Graph drawing aesthetics and the comprehension of uml class diagrams: an empirical study. In *Australian symposium on Information visualisation*, pages 129–137. Australian Computer Society, Inc., 2001. ISBN 0-909925-87-9.

[76] E. Reingold and J. Tilford. Tidier drawing of trees. *IEEE Transactions on Software Engineering*, SE-7(2):223–228, 1981.

[77] S. P. Reiss. An overview of bloom. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–5. ACM Press, 2001. ISBN 1-58113-413-4.

[78] S. P. Reiss. A visual query language for software visualization. In *Proceedings of the IEEE Symposium on Human Computing Languages and Environments*, September 2002.

[79] S. P. Reiss. Visualizing java in action. In *SOFTVIS*, pages 57–65, 210, 2003.

[80] 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.

[81] T. Roxborough and A. Sen. Graph clustering using multiway ratio cut. In G. Di Battista, editor, *Proc. 5th Int. Symp. Graph Drawing, GD*, number 1353, pages 291–296. Springer-Verlag, 18–20 1997. ISBN 3-540-63938-1. URL `citeseer.ist.psu.edu/roxborough97graph.html`.

[82] K. Ryall, J. Marks, and S. M. Shieber. An interactive constraint-based system for drawing graphs. In *ACM Symposium on User Interface Software and Technology*, pages 97–104, 1997.

[83] D. A. Smith, A. Kay, A. Raab, and D. P. Reed. Croquet: A collaboration system architecture, 2003. Whitepaper, http://opencroquet.org.

[84] D. M. Sosnoski. Java performance programming, part 1: Smart object-management saves the day. *JavaWorld*, November 1999.

[85] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern.*, SMC-11(2):109–125, 1981.

[86] K. J. Supowit and E. M. Reingold. The complexity of drawing trees nicely. *Acta Informatica*, 18:377–392, 1983.

[87] I. E. Sutherland. Sketchpad – a man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, May 1963.

[88] K. Van Haaster and D. Hagan. Teaching and learning with bluej: an evaluation of a pedagogical tool. In *Proceedings of the Information Science and Information Technology Education Joint Conference*, June 2004.

[89] P. Ventura, C. Egert, and A. Decker. Ancestor worship in cs1: on the primacy of arrays. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 68–72. ACM Press, 2004. ISBN 1-58113-833-4.

[90] 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. ISBN 1-58113-005-8.

[91] D. West and K. Panesar. Automatic incremental state saving. In *PADS '96: Proceedings of the tenth workshop on Parallel and distributed simulation*, pages 78–85, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7539-X.

[92] M. Wiggins. An overview of program visualization tools and systems. In *ACM-SE 36: Proceedings of the 36th annual Southeast regional conference*, pages 194–200. ACM Press, 1998. ISBN 1-58113-030-9.

[93] M. V. Zelkowitz. Reversible execution. *Communications of the ACM*, 16(9):566, 1973. ISSN 0001-0782.

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