

Sieve: A Java-Based Framework for Collaborative Component Composition

Philip L. Isenhour

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Clifford A. Shaffer, Chairman
Marc Abrams
Dennis Kafura

February 11, 1998
Blacksburg, Virginia

Keywords: Computer-Supported Cooperative Work, Java

Copyright 1998, Philip L. Isenhour

Sieve: A Java-Based Framework for Collaborative Component Composition

Philip L. Isenhour

(ABSTRACT)

This thesis investigates the design objectives for a collaborative workspace based on Sun Microsystems' Java programming language and JavaBeans component architecture. The feasibility of a collaborative component workspace based on these objectives is demonstrated by Sieve, a Java-based framework for collaborative applications. Sieve allows multiple users to collaboratively add, edit, and connect components on a shared two-dimensional workspace.

Sieve introduces a technique that leverages standard JavaBeans mechanisms to support use of "collaboration-unaware" software components. With this technique, components need only conform to basic JavaBeans conventions in order to be shared across collaborating sessions – they need not be programmed specifically for collaboration. Sieve also allows component developers to provide custom mechanisms for sharing components. Sieve is extensible in other ways, allowing developers to introduce new mechanisms for creating, displaying, editing, and connecting components.

Three collaborative applications built on this framework are presented: a visualization environment, a circuit simulation, and a set of tools for composing arbitrary software components. The visualization environment allows construction of dataflow networks from an extensible set of modules. Modules may read data from a variety of sources, filter and transform the data in various ways, and generate visualizations. The circuit simulation allows users to collaboratively construct and analyze simple direct-current circuits. Finally, the "BeanBox Emulation" application reproduces the basic component-linking functionality of Sun's BeanBox builder tool. With this application, users may collaboratively edit and link objects that conform to standard JavaBeans conventions.

This work received support from the National Science Foundation under grant REC-9554206, and the Department of Education's FIPSE program under grant P116B60201.

Acknowledgments

I would like to thank my adviser, Professor Cliff Shaffer, for his support and direction throughout this project. I would also like to thank the other members of my committee, Professors Marc Abrams and Dennis Kafura, for their guidance and suggestions both on this project and on earlier projects which served as a foundation for Sieve.

A number of people contributed to the application-specific components that allow Sieve to do interesting things. Bo Begole and Win Heagy contributed both to the set of visualization components and also to the first Sieve prototype. Jeff Nielsen also implemented several of the dataflow components. Chad Hawthorn wrote the “solver” object used by the circuit simulation, and Adrienne Everett wrote many of the circuit components.

Finally, I would like to express my deepest gratitude to my parents for their unceasing moral support through the course of my studies.

Contents

1	Introduction	1
1.1	Problem Statement	2
1.2	Organization	2
1.3	Terminology	3
2	Related Work	4
2.1	Java and JavaBeans	4
2.2	Collaborative Systems	5
2.2.1	Shared Windowing Systems and Collaboration Toolkits	5
2.2.2	Shared Whiteboards	6
2.2.3	Kansas	6
2.3	Visualization Tools	7
2.3.1	Modular Visualization Environments	7
2.3.2	Web-Based Visualization Systems	7
2.4	Design Patterns	8
3	A Sampling of Sieve-Based Applications	10
3.1	User Interface Support for Collaborative Applications	10
3.2	A Sieve-Based Visualization Environment	12
3.3	A Sieve-Based Circuit Simulation	13
3.4	BeanBox Emulation	14
3.5	Combining Application Components	15

4	The Sieve Prototype and Resulting Objectives	18
4.1	The Initial Prototype	18
4.1.1	Supporting Dataflow Connections	18
4.1.2	Supporting Collaboration	21
4.1.3	Adding a New Display Policy	22
4.2	Requirements and Objectives	22
4.2.1	Flexible Component Construction	23
4.2.2	Flexible Component Linking	24
4.2.3	Flexible Component Selection	25
4.2.4	Flexible Component Display	25
4.2.5	Support for Aggregate Components	26
4.2.6	Minimizing Application Development Overhead	27
4.2.7	Decoupling of Collaboration Support	27
4.2.8	Support for Writing Collaboration-Aware Components	28
5	The Sieve Framework	29
5.1	The <code>Workbench</code> Class	30
5.2	The <code>WorkbenchComponent</code> Interface	32
5.2.1	Types of <code>WorkbenchComponent</code> implementations	33
5.2.2	Component Selection	34
5.2.3	Component Editing	36
5.2.4	Component Movement, Reshaping, and Removal	37
5.3	The <code>Tool</code> Interface	37
5.4	The <code>Link</code> Interface	41
5.5	The <code>WorkbenchListener</code> Interface	48
5.6	Other Sieve Components	49
5.6.1	The <code>WorkbenchComponentEditor</code> Interface	49
5.6.2	Toolboxes	51
5.7	Summary	53

6	Collaboration Support	54
6.1	Distributed Architecture	54
6.2	The Java Shared Data API	55
6.3	Broadcasters, Controllers, and Servers	55
6.3.1	Handling Component Creation	57
6.3.2	Handling Link Creation	58
6.3.3	Handling Other Operations	58
6.4	Component State Replication	58
6.5	Preventing Message “Echo”	60
6.6	Storing Workspace State	62
6.7	Conflict Resolution	63
6.8	The Participant <code>WorkbenchComponent</code> Implementation	65
6.9	The <code>CollaborationSupport</code> class	67
6.10	Summary	68
7	Sieve Application Design	69
7.1	Visualization and Dataflow Component Design	69
7.1.1	Dataflow Network Structure	69
7.1.2	The <code>TableView</code> Interface	70
7.1.3	Supporting <code>TableView</code> Objects in Sieve	78
7.1.4	Adapting <code>TableView</code> Objects to Other Models	79
7.2	Circuit Simulation Design	81
7.3	<code>BeanBox</code> Emulation Design	83
8	Conclusions	89
8.1	Benefits of a JavaBeans-Based Collaborative Workspace	89
8.2	Contributions of the Sieve Design	89
8.3	Future Work	90
8.3.1	Extensions to Sieve Workspace Functionality	90
8.3.2	Extensions to Sieve Collaborative Functionality	92

A UML Overview	96
A.1 Class Diagrams	96
A.2 Interaction Diagrams	97

List of Figures

3.1	Sieve’s whiteboard tools, telepointers, and radar view.	11
3.2	Constructing a dataflow network in the Sieve-based visualization environment. . . .	12
3.3	Two users constructing a simple DC circuit.	14
3.4	BeanBox emulation components used to connect two buttons to an animation bean.	15
3.5	Two visualization components connected using a BeanBox emulation component. . .	16
4.1	Bound properties in the BeanBox.	19
4.2	Event notification in the BeanBox.	20
4.3	Four linked components in a Sieve-based circuit simulation.	24
4.4	A simple simulation component with several sub-components.	26
5.1	High level structure of core Sieve components.	29
5.2	The <code>Workbench</code> class.	31
5.3	The <code>WorkbenchComponent</code> Interface.	33
5.4	Interaction diagram showing <code>WorkbenchComponent</code> selection.	34
5.5	The <code>SelectionListener</code> Interface.	35
5.6	The <code>EditEvent</code> Abstract Class.	37
5.7	The <code>Tool</code> Interface.	38
5.8	The <code>ToolFactory</code> class.	39
5.9	Interaction diagram showing <code>WorkbenchComponent</code> creation by a <code>Tool</code> instance. . . .	40
5.10	The <code>Link</code> Interface.	42
5.11	Interaction diagram showing simple <code>Link</code> creation.	43
5.12	Two linked circuit components in a Sieve-based circuit simulation.	45

5.13	The <code>LinkFactory</code> class.	46
5.14	The <code>WorkbenchListener</code> interface.	48
5.15	The <code>WorkbenchComponentEditor</code> interface.	49
5.16	A component and its property editor.	50
5.17	A hierarchical toolbox.	51
5.18	A toolbar for selecting whiteboard tools.	52
5.19	The <code>ToolListener</code> interface.	52
6.1	High-level interactions between collaborating sessions.	55
6.2	The <code>WorkbenchController</code> interface.	56
6.3	The <code>Message</code> abstract class.	57
6.4	Conflict caused by simultaneous edits.	64
6.5	Resolving conflicting edit messages.	65
6.6	The <code>WorkbenchParticipant</code> class.	66
6.7	The <code>CollaborationSupport</code> class.	67
6.8	Dialog boxes for choosing a channel.	68
7.1	A simple dataflow network.	70
7.2	The <code>TableView</code> interface.	71
7.3	A series of manipulations on a table of data.	73
7.4	The <code>TableViewListener</code> interface.	75
7.5	<code>TableViewEvent</code> hierarchy.	76
7.6	Current <code>TableView</code> filter classes.	77
7.7	Implementation of the <code>TableViewTableViewLink</code> class.	78
7.8	The Swing <code>TableModel</code> interface.	79
7.9	A Swing <code>JTable</code> object used in a dataflow network.	80
7.10	The <code>Solver</code> class.	82
7.11	A button linked to an animation in the <code>BeanBox</code>	84
7.12	A button linked to an animation in <code>Sieve</code>	85
7.13	The <code>BeanBoxComponent</code> interface.	86
7.14	Registering <code>BeanBoxLink</code> with the <code>LinkFactory</code>	87

7.15 The <code>BeanBoxLink</code> implementation, with error checking removed.	88
A.1 UML class diagram.	96
A.2 UML interaction diagram.	97

Chapter 1

Introduction

Component architectures are among the latest developments in the effort to engineer portable, reusable software. These architectures typically consist of application programming interfaces (APIs) and utilities that facilitate the creation of reusable objects and the composition of these objects into larger software systems. Conventionally, the goal of component architectures has been the efficient creation of custom end-user software by assembling and configuring generic, “off-the-shelf” components.

This approach (composition and configuration of reusable components) seems natural, however, for a range of design tasks other than end-user software construction. For example, visualization often involves the design of a set of filters for processing and selecting data. Simple filters can be connected to produce more complex filtering operations, with the resulting data fed directly into visualization components. Image processing applications often follow a similar model. Data from a source image may be passed through a series of simple, linked filters to produce a particular output. Many types of simulations also lend themselves to construction by composition of generic components.

Experiences with shared drawing tools (commonly referred to as “shared whiteboards”) suggest that many types of design tasks could benefit from real-time collaboration. Colleagues might work together to sketch a set of filters for visualization, to produce a rough draft of a simulation design, or to outline the set of components needed to produce a piece of software. A software tool that introduces active components to the traditional concept of a whiteboard adds a new dimension to collaborative design. Instead of working together to produce a static abstraction of a design in words or sketches, colleagues can collaboratively manipulate functional building-blocks to explore possible solutions.

The World Wide Web and Sun Microsystems’ Java programming language [4] provide an ideal foundation for constructing collaborative systems. The popularity of the Web has greatly increased the availability of high-bandwidth networking, making real-time collaboration feasible. In addition to network bandwidth, collaborators must also have compatible software. Java facilitates this by allowing software to be delivered over the web and executed on a variety of platforms without modification. Java also includes component support in the form of the JavaBeans component

architecture. BeanBox, Sun’s reference JavaBeans builder tool, provides an example of a simple environment for editing and connecting beans.

We believe that the JavaBeans architecture holds considerable promise as a foundation for collaborative systems. In addition to promoting realization of many of the promises of the object-oriented paradigm (such as reusability and extensibility), the acceptance of architectures such as JavaBeans should also increase the availability of small, self-contained, reusable software components for a number of problem domains. For example, some of the first beans available were for visualization and database access. As JavaBeans’ popularity increases, beans that perform numerical analysis tasks, provide interfaces to legacy software, and implement common user interface controls will likely become available. As both general purpose and domain specific components are written, a collaborative environment based on a component architecture will allow quick adaptation of these components for collaborative use.

1.1 Problem Statement

This thesis investigates the consequences of a set of modifications made to Sun’s BeanBox to support collaborative visualization. From analysis of these modifications we derive a set of objectives for constructing a more general-purpose collaborative tool for manipulation and composition of software modules. The feasibility of a collaborative component workspace based on these objectives is demonstrated by Sieve, our Java-based framework for collaborative construction of networks of software components.

We present three collaborative applications built on this framework: a visualization environment, a circuit simulation, and a small set of tools for composing arbitrary software components. The visualization environment allows construction of dataflow networks from an extensible set of modules. Modules may read data from a variety of sources, filter and transform the data in various ways, and generate visualizations. The circuit simulation allows users to collaboratively construct and analyze simple direct-current circuits. Finally, the “BeanBox Emulation” application reproduces the basic component-linking functionality of Sun’s BeanBox builder tool. With this application, users may collaboratively edit and link objects that conform to standard JavaBeans conventions.

Our objective is to demonstrate that a component workspace provides a framework for building a variety of interactive collaborative applications. Furthermore, we demonstrate a technique for using standard JavaBeans mechanisms to support use of “collaboration-unaware” software components in a collaborative manner. With this technique, components need only conform to basic JavaBeans conventions in order to be shared across collaborating sessions – they need not be programmed specifically for collaboration.

1.2 Organization

We begin by discussing related systems and technologies in Chapter 2. To provide concrete examples of potential uses for the technology described in this thesis, we describe three Sieve-based applications in Chapter 3.

In Chapter 4, we discuss our experiences with the initial prototype of Sieve, and the set of objectives that we extracted from these experiences. The design for the core components of the Sieve workspace is presented in Chapter 5, and the design for collaboration support mechanisms is described in Chapter 6. In Chapter 7 we return to the three applications presented in Chapter 3, describing their implementation in greater detail.

1.3 Terminology

In our discussions of Sieve and Sieve-based applications, we differentiate between three categories of objects:

1. **Core Sieve framework components that provide basic workspace and collaborative functionality.** These objects provide support for adding, removing, editing, and linking objects on the workspace, and for replicating the workspace between collaborative sessions. Our design of these components is the most significant part of this work, and is described in Chapters 5 and 6.
2. **Application-specific components that provide tools for a given problem domain.** Examples include components for visualizing data and components for constructing simulations. These components may be written specifically for Sieve, or may be completely independent of Sieve. Components for our three example applications are described at a high level in Chapter 3. Lower-level design of these components is then presented in Chapter 7, after we have presented the design of the framework components.
3. **Objects that provide the user interface for accessing the Sieve workspace and selecting application-specific components.** The design of these objects can vary widely depending on how the core components are used in the context of the larger piece of software that a user interacts with. Therefore, we generally do not address the design of these objects in detail.

It may be useful for the reader to consider that under this decomposition, Sieve is somewhat analogous to Sun's HotJava bean [32]. This bean provides only the basic functionality for rendering HTML and similar data. It relies on external content (in the form of web pages) to make it do something useful, and on external code to provide a user interface for pointing it at content and showing it to the user. Similarly, Sieve provides core functionality for a shared workspace, while application-specific "content" components allow Sieve to do something useful for a given domain of problems, and user interface objects provide access to the workspace and to the application modules.

Chapter 2

Related Work

This chapter reviews numerous existing and emerging technologies that are related to the work presented in this thesis. We begin by discussing the Java programming language and the JavaBeans component architecture. We then describe existing collaborative systems that have influenced our design. Since Sieve began primarily as a visualization environment, we also examine related visualization systems. Finally, we give a brief overview of a set of object-oriented design patterns that are used in Sieve’s design.

2.1 Java and JavaBeans

Sun’s Java programming language [4] is an object-oriented language with syntax similar to that of C++. Java is platform neutral, in that it can be compiled into bytecodes that can then be interpreted by “virtual machines” on many different platforms and operating systems. The inclusion of Java virtual machines in popular web browsers has led to widespread use of Java as a programming language for active content on the World Wide Web. Small Java programs, or “applets,” can be embedded in Web pages. Java can also be used for larger, stand-alone applications.

Standard Java libraries includes extensive support for networking and distributed computation. TCP and UDP sockets are available, as is a remote method invocation mechanism. The latter simplifies interactions among Java objects on separate machines. Standard libraries also include support for building platform-independent user interfaces, streams-based access to both local and remote files, and serialization of objects for storage or transmission.

Sun’s JavaBeans [31] component architecture is a set of APIs intended to support the creation of Java components that can be composed together into applications by end users. The three most important features of the JavaBeans architecture are the ability to discover (and use) properties exposed by an object, events generated by an object, and methods exported by an object. A property is a named attribute that may be queried and modified by invoking methods that conform to a particular naming pattern. Events are objects that are passed from one object to another to provide notification when something interesting has happened. An exported method is simply a method that has been marked as visible to other objects. Within a builder tool, the exported

methods of an object may be invoked in any way that the tool supports. Builder tools may, for example, allow invocation of an exported method when an event is fired by another object. To support builder tools, the JavaBeans package provides the ability to inspect (or “introspect,” in JavaBeans terminology) arbitrary objects to determine the properties, events, and methods they expose.

Sieve both relies on and extends JavaBeans functionality. The original prototype of Sieve was a modified version of Sun’s reference implementation of a JavaBeans-based builder tool, the BeanBox. The current implementation of Sieve retains much of the feel and functionality of BeanBox.

In the months since the first JavaBeans implementation was released by Sun, numerous other vendors have released JavaBeans-compliant builder tools. IBM, Borland, and Symantec have adapted existing products to take advantage of the JavaBeans standard, and new tools such as Penumbra Software’s SuperMojo [24] have been developed specifically to support JavaBeans.

The most obvious way in which Sieve differs from other JavaBeans-based builder tools is its integrated support for synchronous collaboration. Furthermore, it allows applications built on Sieve to support much richer forms of interaction between components than those defined in the JavaBeans specification and implemented in the BeanBox. This primarily serves to simplify the user interface of a Sieve-based application, as it allows multiple primitive linking operations to be combined into a single operation.

2.2 Collaborative Systems

Much of the design for Sieve’s collaboration support builds on ideas from numerous earlier synchronous collaborative systems. Synchronous collaboration refers to the ability for multiple collaborators to manipulate shared artifacts at the same time. The current generation of multi-player, real-time, dungeon-style adventure games are examples of synchronous collaborative systems. With asynchronous collaboration, users may still manipulate shared artifacts but may not do so at exactly the same time. E-mail and bulletin board systems in which multiple users may read and post messages are examples of simple asynchronous collaborative systems.

2.2.1 Shared Windowing Systems and Collaboration Toolkits

Traditional approaches to constructing collaborative applications typically fall into one of two categories. Collaboration transparency systems make existing single-user applications collaborative by using shared windowing mechanisms to provide multiple users with simultaneous access [21]. Java Applets Made Multi-user (JAMM) [6] provides shared windowing functionality for Java applets. Microsoft’s NetMeeting allows collaborative use of many Windows-based applications [23]. Hewlett Packard’s SharedX product provides similar functionality for UNIX machines running X-Windows [17]. Lauwers and Lantz discuss the shortcomings of SharedX and similar windowing systems with respect to use for application sharing [22].

Collaboration aware systems are constructed with the explicit intention of being used collaboratively. The chief drawback of these systems is the extra effort required to design for collaboration.

Although numerous toolkits for constructing these applications have been developed (for example, [8], [18] and [13]), the overhead of learning and using these toolkits may be too great a hindrance to the development of large numbers of components for different problem domains.

Sieve's approach to collaboration support lies somewhere between these two extremes. It does not provide true collaboration transparency, but it also does not require developers to use a collaboration toolkit. Instead, it requires that shared components conform to the conventions of the JavaBeans component architecture. Specifically, the shared components must expose their state as bound properties. There is, of course, overhead in learning the JavaBeans APIs and conventions. However, this knowledge is more generally applicable than knowledge of a specific collaboration toolkit.

2.2.2 Shared Whiteboards

The software artifacts we describe in this work have a great deal in common with collaborative drawing tools, or "shared whiteboards."

Numerous shared whiteboard systems have been implemented and their use studied in considerable detail. Early examples include a study of the GroupSketch and GroupDraw systems by Greenberg, et al. [12] Analysis of the fundamental ways in which groups use shared drawing spaces (computer-supported or traditional) has been conducted by Tang [38]. Tang's work identifies both the basic functions of a shared drawing space (storing information, expressing ideas, and mediating interaction) as well as the basic actions of users of such a space (listing, drawing, and gesturing).

We have implemented components that support these basic actions on the Sieve workspace. Shared text areas may be used for posting notes and labels, and simple drawing tools are provided. Gesturing is supported through telepointers. In addition to these actions, Sieve provides the opportunity for new kinds of action: composition and editing of active components.

2.2.3 Kansas

Kansas [28] is a collaborative system that pioneered the concept of the collaborative workspace. It provides a large, flat surface that may be inhabited by multiple users. Users have a window into a rectangular portion of the space, and may move this window to work either in isolation or in the same part of the space as other users.

Components that exist in this space are Self objects. Self is a prototype-based (as opposed to class-based) object-oriented language. One of the uses for Kansas is, in fact, as a collaborative development tool for the Self language. Users can edit objects on the Kansas workspace to add, remove, or change methods and attributes. In practice, the types of objects commonly used in Kansas include interactive simulations, user interface components, and communication tools such as live video images.

2.3 Visualization Tools

Although Sieve is not simply a visualization tool, visualization is certainly one of its most natural applications. As such, it has at least some similarity to a wide range of existing visualization tools.

2.3.1 Modular Visualization Environments

Modular visualization environments (MVEs) such as AVS [39], Data Explorer [1], and apE [9] present the user with a set of modules representing sources of data, data manipulation processes, and image generation tools. These modules can then be linked to specify the flow of data to create a “program” for producing a visualization. Some MVEs also provide APIs that allow new modules to be written by end users. The success of these packages has demonstrated the power of direct manipulation of dataflow networks as a user interface mechanism.

The visualization environment we have implemented with Sieve (described in Section 3.2) borrows many of the dataflow network ideas first implemented in these MVEs.

2.3.2 Web-Based Visualization Systems

The explosion in popularity of the Web has opened opportunities for distributing both raw data and visualization tools to a wider user population. This potential is being realized in a number of forms.

Systems have been developed that provide an HTML forms interface to drive visualization software on a remote server. For example, Wood et al. describe such a system for visualizing environmental data through a web browser [41]. This approach gives users on low-end machines access to powerful software through the Web, but limits the degree of interaction. In particular, direct-manipulation interfaces cannot be built using HTML forms.

Similar systems even attempt to support collaborative analysis of visualizations. The CoVis project [7], for example, has developed tools to support scientific visualization “in a collaborative context” for use primarily in K-12 science education. These tools allow visualizations to be accessed over the web by students communicating with teleconferencing software, but the visualizations themselves are not manipulated collaboratively.

Applets for generating and delivering visualizations over the Web were among the first tools written in Java, and numerous packages are available that allow software developers to generate charts and graphs within Java applets and applications. For example, Alper, et al. [3] describe an applet that allows exploration of geospatial datasets. Data are retrieved from a remote server and the visual output is produced locally by the applet. Visual Numerics’ suite of JWAVE applets communicate with a server that generates visualizations remotely [40]. In this case a Java applet is used to drive the remote (non-Java) software and display the resulting image.

Javamatic [25] generalizes the idea of providing Java-based interfaces for software running on remote servers. It supports automatic generation of applets that provide user interfaces to arbitrary

command-line driven legacy applications. The output of a remote execution of such an application is then made available over the web for viewing or further manipulation.

The use of interactive applets represents a significant step forward from earlier batch-mode, forms-based interfaces to visualization tools. In addition to special-purpose tools for communicating with remote servers, an increasing number of simple visualization applets for charting and graphing have been made available on the Web. These applets represent a potentially valuable source of module implementations for Java-based visualization environments.

2.4 Design Patterns

Object-oriented design patterns are descriptions of high-level solutions to common design problems [11]. They provide general descriptions of compositions of communicating objects. These general compositions can be customized to solve specific problems.

The terminology of design patterns appears throughout our design descriptions. A few of the patterns documented by Gamma et al. are particularly common in our work. We introduce those briefly here.

Abstract Factories encapsulate the logic for creating objects. They allow a client object to request that an instance of one of a family of classes be instantiated. The advantage of factories is that they isolate the client from knowledge of specific concrete classes. The client need only have knowledge of an abstract superclass (or, in Java, an interface). This makes it easier to add or change the available concrete classes, since no changes to any client objects are needed.

Java's Abstract Windowing Toolkit (AWT), for example, provides a factory for constructing basic interface elements. A call to a `java.awt.Toolkit` instance's `createWindow` method always returns an instance of a `WindowPeer` subclass representing a platform-dependent window object. The logic for determining which specific concrete subclass to instantiate is hidden from the client object.

Builder objects also encapsulate the logic for creating objects. The intent of a builder is to separate the process of constructing a complex object from the object's representation. This approach allows the same builder object to create different types of objects that share the same construction process.

Consider, for example, a vector drawing tool. Objects representing lines, rectangles, and ovals can all be constructed based on two pieces of information: the point at which the user started to drag the mouse and the point at which the user released the mouse. A builder object could be implemented that detected these two points and used them to construct a line, a rectangle, or an oval.

Singleton objects are objects that have only a single instance and that provide a well-known mechanism for accessing that instance. For example, a single instance of `java.awt.Toolkit` exists in a Java virtual machine. A reference to it may be obtained by a client object by calling the static method `getDefaultToolkit()`.

A **Facade** provides a simple interface to a complex subsystem, making the subsystem easy to use. Facades allow a client to use a simple interface when possible, while allowing direct access to parts

of the subsystem when needed. Borrowing Gamma et al.'s example, a compiler subsystem may include many small objects such as a scanner, a parser, a code generator, and others. Most clients will likely just need to give the compiler an object containing source code, without having any knowledge of the subsystem's parts. A facade could provide an interface to the compiler, hiding the details of instantiating and connecting the parts. More sophisticated clients might, however, need to bypass the facade and access the subsystem parts directly.

Command objects encapsulate requests. Concrete implementations of an abstract command can then perform different actions. For example, a button object can be parameterized with a command object such that the command is invoked when the button is pressed. The specific action triggered by the button can then be changed by giving it a different type of command object.

Memento objects capture and externalize an object's internal state so that the object can be restored to this state later. An "undo" mechanism, for example, can make use of mementos. Before any change to an object, the state of the object can be captured as a memento. If it becomes necessary to undo the change, the memento can be used to restore the previous state.

An **Observer** object is notified when the state of another object changes. This pattern is the foundation for Java's delegation event model: Listeners may be registered with user interface (and other) objects, and notified when something interesting happens. For example, listeners may be registered with Java window objects. When the window is hidden, exposed, or minimized, all registered listener objects are notified.

Numerous other patterns have been documented (see [11] and [26]), but those listed above are among the most prevalent, both in the design of the system described in this thesis and in standard Java libraries.

Chapter 3

A Sampling of Sieve-Based Applications

Sieve has been designed to support a wide range of collaborative design activities. In this chapter we describe three sets of application components that have been developed or adapted for use with Sieve. The focus of this thesis is on the “core” components of the Sieve framework, and how the core components may be used by application developers. In this chapter, however, we describe application components as they appear to a Sieve user. This serves as orientation for the description of the Sieve framework in Chapter 5 and collaboration support in Chapter 6. The implementation of specific application components described in this chapter will be presented in Chapter 7.

Sieve is intended as a framework for applications whose primary intent is to allow the user to edit and connect components on a two-dimensional workspace. Components may be arbitrary JavaBeans, in which case Sieve treats them as “black boxes,” using only Java’s introspection mechanism to determine how they may be edited. Components may also be written specifically for Sieve, giving the application writer more control over the interaction between the Sieve framework and the application components. Connections may represent any kind of application-defined association between components.

3.1 User Interface Support for Collaborative Applications

All Sieve-based applications make use of a scrolling, two-dimensional workspace. Sieve supports flexible collaboration within the workspace and provides real-time information about participants’ actions and locations. A range of collaboration styles are supported by providing location-relaxed WYSIWIS (What You See Is What I See) [29], where collaborators may view and manipulate the same or different parts of the shared data simultaneously. Hence while changes made to the workspace are propagated to all remote collaborators, all collaborators need not be working in the same part of the workspace simultaneously, since not all collaborators need to have the same portion of the workspace within their current view.

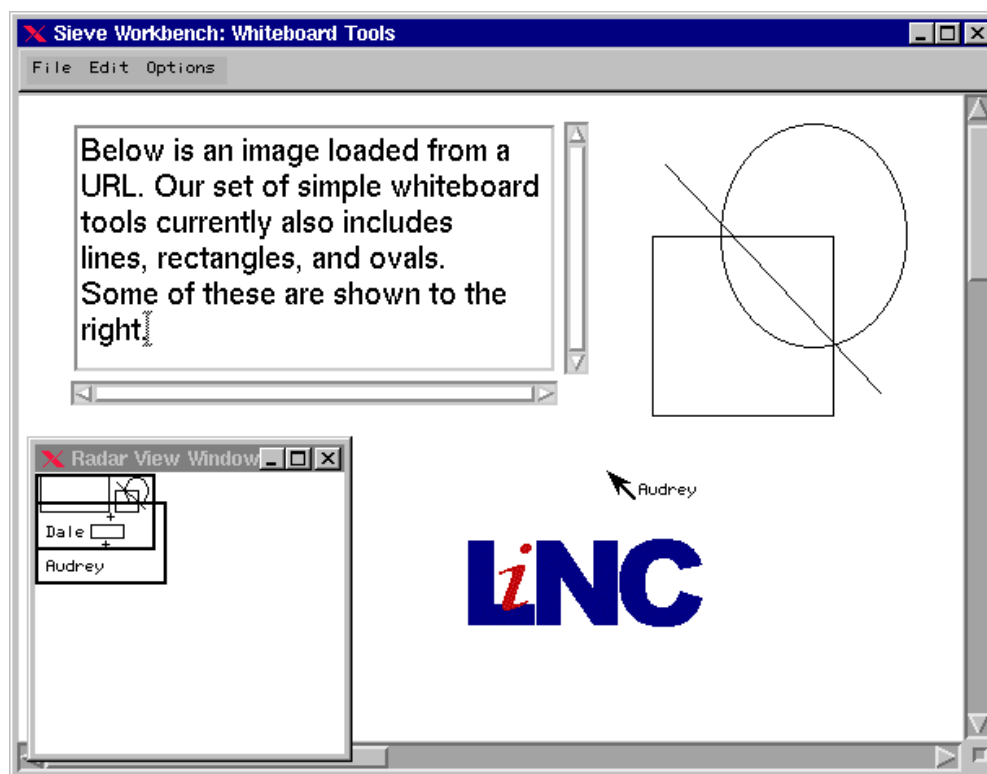


Figure 3.1: Sieve’s whiteboard tools, telepointers, and radar view.

Workspace awareness (continuous knowledge of remote participants’ interactions and locations [15]) is provided through two interface elements: telepointers and a multi-user overview of the workspace. A telepointer represents a remote user’s mouse pointer position and thus provides location awareness. Collaborators can also use telepointers to gesture at items on the workspace to augment communication. To provide additional workspace awareness information, Sieve uses a multi-user overview, or radar view [15], of the workspace. The radar view displays a rectangle for each user, representing that user’s viewport into the workspace and providing additional location information. Remote users’ mouse positions are also indicated on the radar view.

The state of each Sieve session is stored on the server, allowing “late-joiners” to be brought up to date. This persistence mechanism also allows Sieve to be used for *asynchronous* collaboration. Collaborators may work at different times, leaving their modifications for coworkers to manipulate later. The design of the persistence mechanism is described in Section 6.6.

Sieve additionally supports asynchronous collaboration by providing a set of whiteboard-style tools for annotating the workspace. Text, images, lines, and other shapes can share the workspace with application-specific components. These tools enhance Sieve’s support for asynchronous collaboration, since one collaborator can leave notes on the workspace for later review by other collaborators.

Figure 3.1 shows two users (Dale and Audrey) collaborating in a Sieve workspace. The radar view in the lower left corner shows each user’s viewport. The location of Audrey’s pointer is indicated

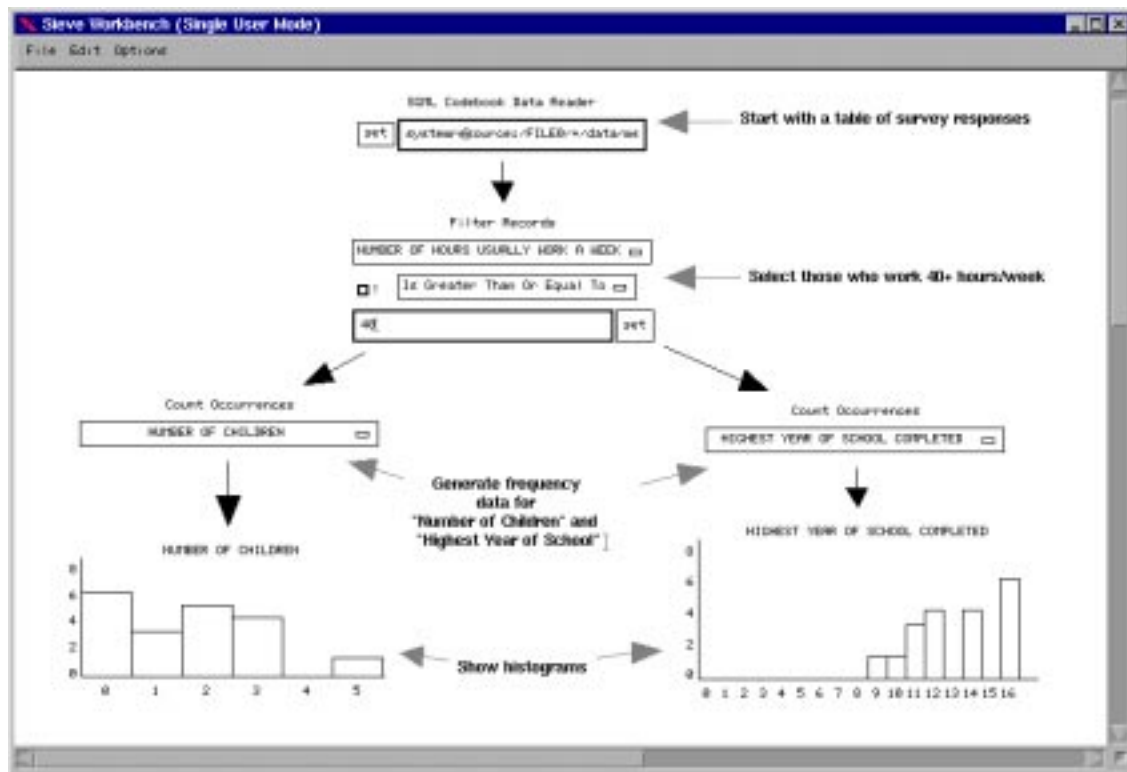


Figure 3.2: Constructing a dataflow network in the Sieve-based visualization environment.

by a telepointer labeled with her name. Sieve’s whiteboard tools have been used to add a text box, an image, and some lines and shapes to the workspace.

Telepointer and radar view support, workspace persistence, and whiteboard tools are available to all Sieve-based applications.

3.2 A Sieve-Based Visualization Environment

The first application developed for Sieve was a collaborative visualization environment.¹ This application presents the user with a large, scrollable workspace onto which data sources, filtering modules, and visualization components may be dropped, linked, and configured. Our design allows processing and visualization modules to be generic, with all data-source specific details hidden by the source modules. All dataflow modules implement a Java interface that allows data to be retrieved by adjacent modules in the network as a two-dimensional table containing objects of any type supported by Java. This interface is called a **TableView**.

¹The name “Sieve” was originally derived from the intent of this application: “Collaborative Interactive Visualization,” or “CIV”.

Modules representing data sources can be written for a wide range of raw data formats and sources. These may include objects that parse files retrieved over the Web, objects that access SQL or other databases, and objects that retrieve data from remote servers using CORBA, Java’s Remote Method Invocation library, or proprietary protocols. Once raw data have been retrieved by a source module, processing modules can manipulate these data and present an altered or extended `TableView`. Visualization modules then produce visual representations of the data in a `TableView`. Visualization modules can also serve as an interface for data selection, in which case they may present an altered or extended `TableView` to adjacent modules.

The resulting dataflow network is fully interactive, using an event mechanism to notify interested modules of changes to the data or to the configuration of the network. When an event is received, modules can retrieve new or modified data from their source. Users modify the network directly on the workspace, with changes to both the structure of the network and the modules themselves reflected to all collaborators as quickly as processing power and network speed permit. The design of the dataflow components is described in greater detail in Section 7.1.

Figure 3.2 shows a Sieve workspace containing a simple dataflow network. The user is exploring a subset of the 1993 General Social Survey (GSS) [14] data by selecting only those responses from people who worked 40 or more hours per week, counting the occurrences of values for the “Number of Children” and “Highest Year of School Completed” variables, and then generating bar charts. The steps have been labeled on the workspace using the annotation tools described in Section 3.1.

3.3 A Sieve-Based Circuit Simulation

Figure 3.3 shows two users collaboratively constructing a simulation of an electric circuit. The circuit simulation application consists of a palette object (in the upper-left quadrant of the workspace) and a set of components representing basic direct-current electric circuit elements: batteries, resistors, lightbulbs, ammeters, voltmeters, and terminals.

In the example, the two users have constructed a simple circuit consisting of a battery, three resistors, and an ammeter. As with the visualization environment, location awareness is provided by a radar view and telepointers.

Components can be added to the workspace by dragging them off of the palette. Users may select a component for editing by clicking on it. In this case, the user has selected the right-most resistor. The property panel (in the upper-right quadrant of the screen shot) can be used to change the resistance of this component. The circuit is “live” in the sense that a change to a component or to the structure of the circuit causes an immediate reaction. Editing the resistance value, for example, would cause the value displayed by the ammeter to be updated.

The circuit simulation components support a different style of user interaction than that supported by the visualization components. For example, circuit components typically have multiple terminals to which wires can be connected. With our visualization components, links are made from one component to another. With the circuit components, links are made from a terminal within one component to a terminal within another component. This difference illustrates Sieve’s ability to support alternate interface mechanisms for different applications. Dataflow network construction

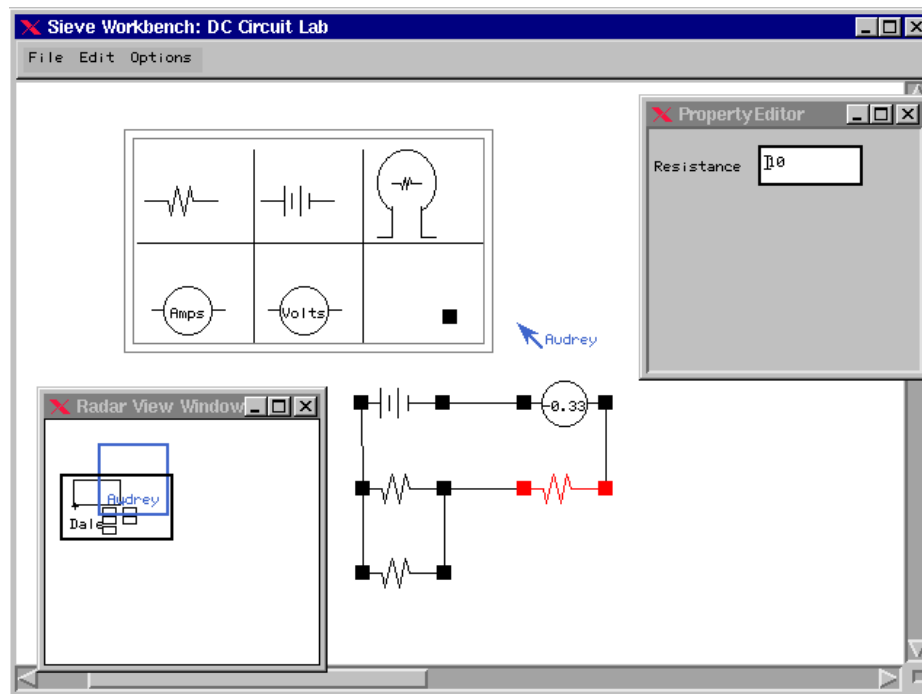


Figure 3.3: Two users constructing a simple DC circuit.

requires relatively course-grained, module-to-module linking. Circuit construction requires a finer granularity of linking.

The design of the circuit components is described in Section 7.2.

3.4 BeanBox Emulation

As we discuss in Chapter 4, the first prototype of Sieve was produced by modifying Sun's BeanBox builder tool. The current version of Sieve preserves the basic component composition mechanisms provided by BeanBox, but implements them in a different way. Instead of BeanBox's invisible connections between objects, Sieve includes adapter components that can be used to link arbitrary objects on the Sieve workspace.

Figure 3.4 shows how several of the sample beans included with the BeanBox distribution can be connected on the Sieve workspace. Two buttons ("Start Animation" and "Stop Animation") are linked to a "juggler" animation component using two instances of our "Event Adapter" component. The adapter component lets the Sieve user select any event fired by the adapter's source component. The selected event can then be linked to a compatible method in the adapter's destination component.

In this example, the `actionPerformed` event – the event generated when a Java button is pressed and released – is linked to the `start` and `stop` methods of the animation component.

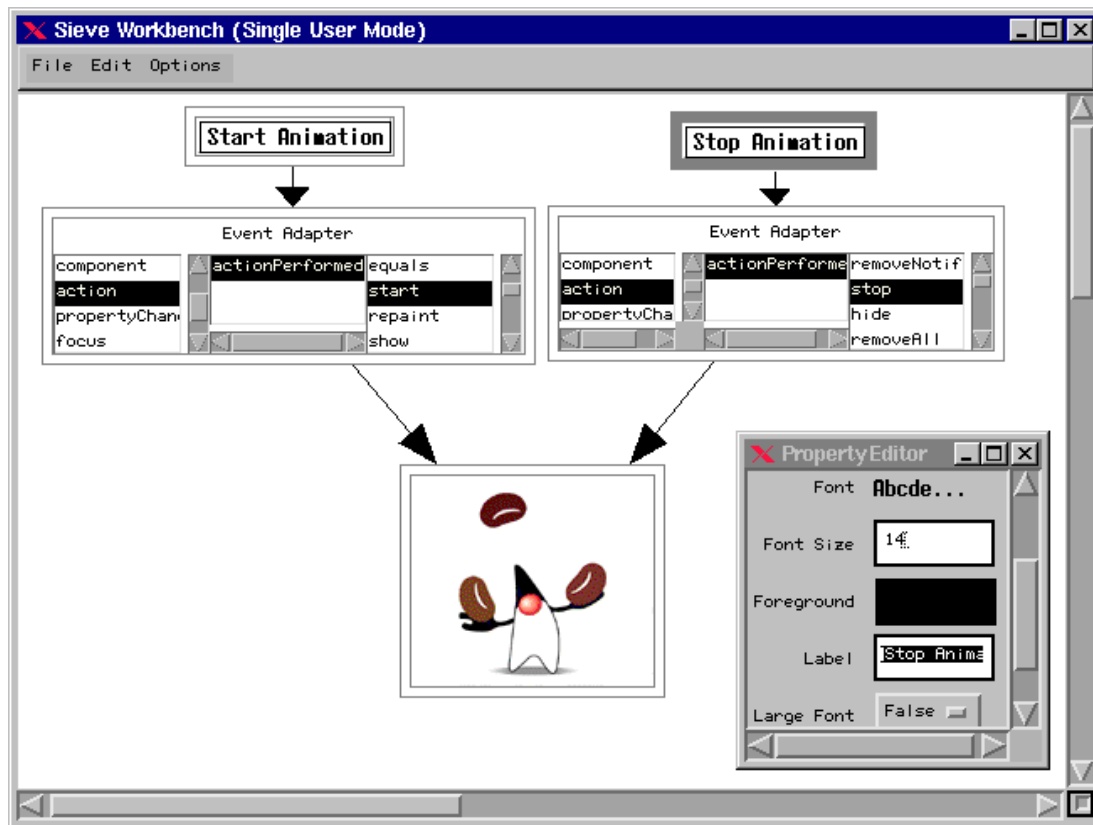


Figure 3.4: BeanBox emulation components used to connect two buttons to an animation bean.

We have implemented this set of BeanBox emulation tools primarily as a proof of concept, illustrating that the basic object composition functionality of BeanBox is preserved in Sieve. However, tools supporting collaborative composition of beans could be quite useful. The intent of such a tool would likely be somewhat different from that of the BeanBox, focusing on support for collaborative design and brainstorming rather than on creating software for distribution. It would, however, certainly be possible to provide some means for hiding links, adapters, and invisible beans, as well as for generating a distributable, standalone version of the workspace content as a Java applet or application.

The design of the BeanBox emulation tools is described in more detail in Section 7.3.

3.5 Combining Application Components

All application components described in this chapter are, from the perspective of the Sieve workspace, simply content. The workspace implementation has no knowledge of any application-specific classes. As such, components from multiple applications can exist on the same workspace at the same time.

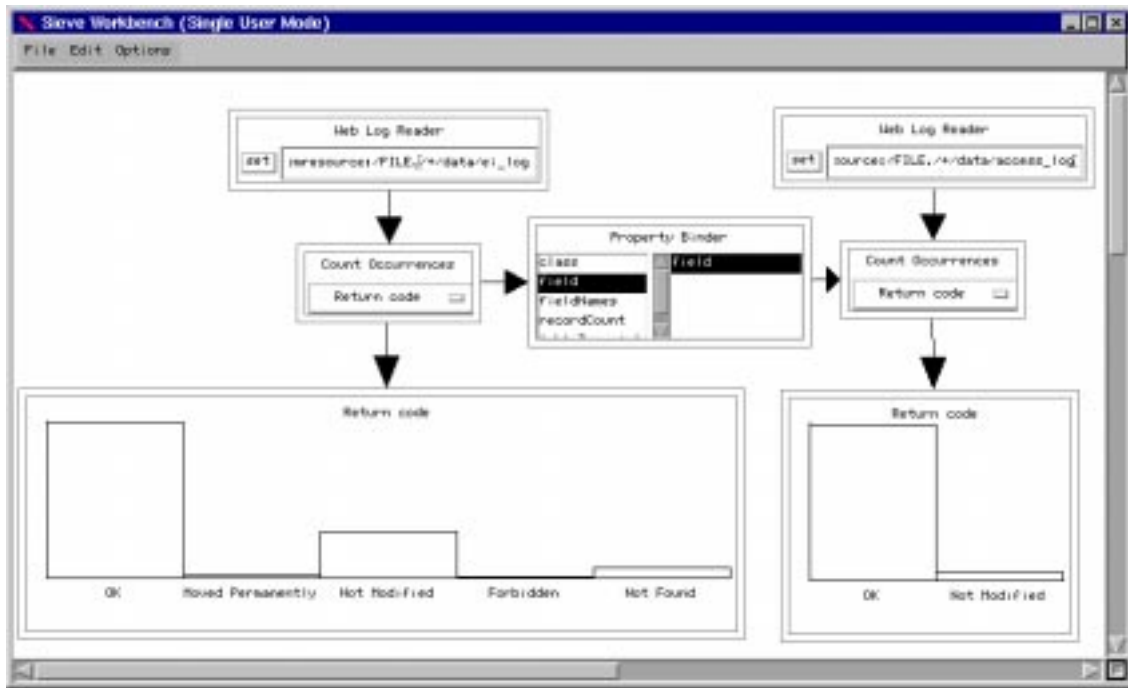


Figure 3.5: Two visualization components connected using a BeanBox emulation component.

Our set of whiteboard tools, for example, may be thought of as constituting a simple application. These components may be used on the Sieve workspace along with any other kind of components. They may therefore be used to annotate dataflow networks, circuits, BeanBox-style compositions, or any other workspace content.

The distinction between applications is also somewhat arbitrary. From the perspective of Sieve's BeanBox emulation tools, the components used in the visualization environment are simply beans. Figure 3.5 illustrates this idea. Two “Web Log Reader” components are providing data from the log files produced by web servers. Data from each of the two readers are then passed to a “Count Occurrences” component, which generates frequency data for a given field in the data. The frequency data are then displayed on a histogram. The two “Count Occurrences” components are linked using one of our BeanBox emulation tools: a property binder. The “field” property of the component on the left is bound to the “field” property of the component on the right. Hence when a new field is selected in the component on the left, the change is propagated to the component on the right.

As another example, consider the visualization and circuit components. There is currently no meaningful way to connect a visualization component to a circuit component, since circuit components do not generate tables of data and visualization components do not have electrical characteristics such as resistance. The two sets of components were designed independently of one another, and do not share any code aside from classes found in the standard Java libraries.

It is, however, conceivable that a component could be written that act both as a part of an electric circuit and as a part of a dataflow network. Currently, our set of circuit components only includes components whose behavior does not change over time. We therefore do not have components such as capacitors, inductors, or semiconductors. If these components were added, it would be desirable to add a special kind of meter component that generates a table of current measurements at regular time intervals. If properly designed, the meter could be linked into the circuit like any other circuit component, but could also be linked to visualization and analysis modules.

Chapter 4

The Sieve Prototype and Resulting Objectives

This chapter provides a brief history of Sieve’s development, followed by a discussion of the issues raised by early Sieve prototypes. From these issues we have extracted a set of objectives for the design of a collaborative component workspace. The design of our current version of Sieve (described in Chapters 5 and 6) addresses these objectives.

As discussed in Section 1.3, we use the term “application” to describe a set of components used to explore a problem in a given domain. For example, we consider the set of dataflow and visualization components used in the visualization system described in Section 3.2 to be an application. In this context, the term “component” refers to an object in the Sieve workspace. These objects may or may not be instances of the Java user interface class `java.awt.Component`.

4.1 The Initial Prototype

The first Sieve prototype was constructed by modifying Sun’s BeanBox builder tool to support collaborative visualization. Our goal was to produce a Java-based collaborative modular visualization environment (MVE), with functionality similar to existing MVEs such as AVS [39]. BeanBox provided an attractive starting point, since it supported the idea of a workspace onto which arbitrary objects could be placed and manipulated.

The modifications that we made to the BeanBox introduced support for dataflow connections, added collaboration support, and altered the way that components were displayed and edited. In this section we describe these changes in greater detail.

4.1.1 Supporting Dataflow Connections

The modifications made support creating dataflow networks overcame limitations in the two mechanisms for connecting components in the BeanBox: bound properties and event notification.

Property Binding

The BeanBox uses property binding to connect two beans by linking a property of one bean to a similarly-typed property in another bean. When the property of the first bean changes, the bound property in the second bean will be set to the new value.

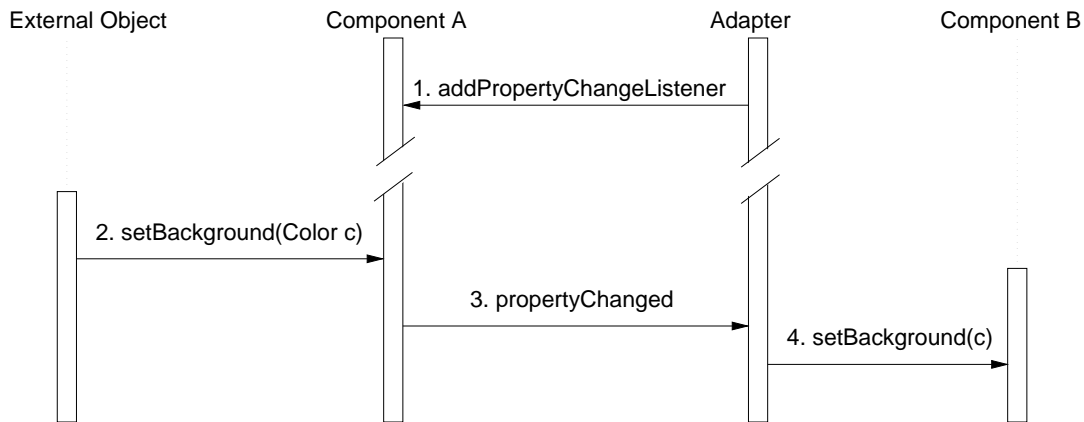


Figure 4.1: Bound properties in the BeanBox.

Figure 4.1 shows how bound properties are implemented in the BeanBox. In this example, the background colors of two components (“Component A” and “Component B”) are bound to each other. The object interactions are as follows:

1. When the properties of the two components are first bound, an adapter object is generated. This object implements the JavaBeans `PropertyChangeListener` interface, and is registered with Component A by calling A’s `addPropertyChangeListener` method and passing the adapter object as a parameter.
2. Something changes Component A’s background color by calling A’s `setBackground` method. The change could be initiated by an external object such as a BeanBox-style property editor, or by another bean. Component A could also invoke `setBackground` on itself, perhaps in response to some user action.
3. Component A changes its background color and then fires a property change event. This results in the Adapter’s `propertyChanged` method being invoked.
4. The Adapter will check the name of the property that has changed. If (as in this example) it is a property that is bound to another component, it will change that component’s background by calling `setBackground`.

In BeanBox, a user initiates property binding by selecting the “source” component (Component A in the previous example) and selecting “Bind Property…” from the Edit menu. This produces a dialog from which the user may select one of the source component’s properties. Next the user selects the destination component (Component B in our example). This produces another dialog

that shows a list of compatible properties. For example, if the user first selected a String property in the source component, only String properties in the destination component would be listed. Once a property is selected, an adapter object is constructed and the scenario shown in Figure 4.1 begins.

Event Notification

BeanBox’s event notification mechanism allows two beans to be connected such that when a particular event is generated by one bean, a specific method is invoked on the other.

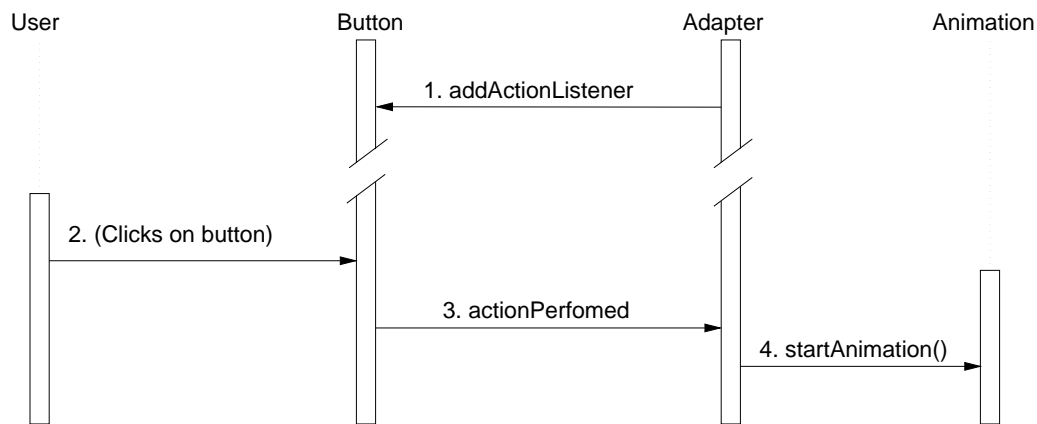


Figure 4.2: Event notification in the BeanBox.

Figure 4.2 shows how a button may be bound to an animation object. The intent is that clicking the button generates an event that results in the invocation of a method that starts the animation. In this scenario:

1. When the connection is first established, an adapter object is generated that implements the AWT `ActionListener` interface. This object is registered with the Button.
2. At some point during execution, the user clicks the Button.
3. The Button generates an `ActionEvent`. This results in the Adapter’s `actionPerformed` method being invoked.
4. The Adapter invokes the Animation’s `startAnimation` method.

BeanBox’s user interface for establishing this kind of binding of an event to a method is similar to its interface for property binding. The user selects the source component, and then chooses an event name from a menu. The user can then select the destination component. BeanBox presents a dialog showing all compatible methods in the destination object. When the user selects a method, the adapter object is generated.

The property binding and event notification mechanisms are similar in that in both cases, an event generated by a source object is received by an intermediate adapter object. The adapter object then invokes a specific method on the destination object. Property binding is, in essence, just a special case of the event notification mechanism.

Dataflow Connections

In their BeanBox implementations, neither property binding nor event notification provides a natural way to connect the dataflow modules used in our Sieve-based visualization environment. As with property binding and event notification, links between these components are directional. When two dataflow components are connected, one acts as a source of data and the other acts as a consumer. These components require two channels of communication: one to “push” events when something changes, and the other to “pull” data when first connected or when an interesting change occurs. When a link is created, the destination component must be registered with its source as an event listener, so that the destination component can be notified of any changes to the structure or content of the data. The destination component must also be given a reference to its source. This allows the destination component to invoke methods on its source to retrieve (or “pull”) data whenever it is needed. It is this kind of connection (giving the destination a reference to its source) that neither the bound property mechanism nor the simple event notification mechanism can easily provide. (The design of the dataflow components is explained in more detail in Section 7.1.2.)

4.1.2 Supporting Collaboration

Support for synchronous collaboration requires propagating both the effects of user manipulations of beans in the BeanBox and also the movements of the user’s mouse pointer. Our modified BeanBox generated messages when a user added, removed, edited, connected, or disconnected components. These messages were then sent to other collaborating sessions, where the actions described by each message were repeated.

A variation of the property binding mechanism described above was used to propagate the effects of user manipulation of beans on the workspace. Property change listeners were registered with each visualization component upon creation. When a property change was detected, a message was generated that contained a component identifier, the property name, and the new value of the property. This message was sent to the other sessions, who used Java’s introspection mechanism to find and invoke the appropriate method to reproduce the property change. This approach allowed any component state that was exposed as a bound property to be replicated across all collaborating sessions.

To support activity awareness, information about each user’s mouse pointer location was used to draw a telepointer on the BeanBox. To support relaxed WYSIWIS, a multi-user radar view was added. This component (described in Section 3.1) displayed a miniature view of the BeanBox with indications of each user’s viewport and mouse location.

4.1.3 Adding a New Display Policy

We also modified the way that “invisible” beans are displayed in the BeanBox. In the BeanBox, beans that are inherently visual (i.e., those that are subclasses of `java.awt.Component`) are displayed on the workspace. Invisible beans are simply displayed as a small rectangle containing the name of the bean’s class.

We changed this display policy to better accommodate modifying modules in a dataflow network. Since modules representing data sources and filters are inherently invisible, we used the two types of editors supported by the JavaBeans specification (property editors and customizers) to provide a visual representation for otherwise invisible components.

Property editors are simple GUI components that are constructed at run-time based on introspection of a bean’s properties. For each property that a bean exposes, a builder tool may present an editor for changing the value of that property. For example, a color-picker widget may be displayed to allow editing of properties that are colors.

Customizers are optional, self-contained graphical interfaces for customizing the appearance or behavior of a bean. Unlike property editors, customizers are not generated by builder tools. Instead, they are implemented specifically for a particular kind of bean. While this introduces additional overhead for the bean writer, it allows for the implementation of much more complex interfaces than those provided by automatically-generated property editors.

The BeanBox presents a property editor in a separate window for whatever bean is currently selected. The customizer for a given bean, if it exists, is available by selecting a menu option. Our modifications simply placed the appropriate type of editor directly on the workspace. This allowed the behavior of otherwise invisible filters to be more easily determined when viewing a dataflow network.

For example, consider the dataflow network shown in Figure 3.2. The components labeled “Filter Records” and “Count Occurrences” represent objects that, when used outside of a tool such as Sieve or BeanBox, have no need for a graphical interface. The classes that implement these filters are therefore not subclasses of `java.awt.Component`, making them invisible. The interface objects shown in this figure are customizers for these components.

4.2 Requirements and Objectives

Modification of the BeanBox allowed the prototype to be developed rapidly, but also revealed a number of limitations. The BeanBox was, quite simply, not designed to be extended in all of the ways that our application required. A few of the desired extensions could be made by subclassing BeanBox classes or by composing BeanBox classes with new classes. For example, it was possible to use the delegation event model of Java’s Abstract Windowing Toolkit to attach listeners to detect the user’s mouse movements for propagating telepointer events. Most other extensions, however, required modifying the BeanBox code.

In nearly all cases, the limitations of our modified BeanBox were simply problems with flexibility. BeanBox provided *almost* all of the functionality that our application required. However, each new application would require modifications to the BeanBox itself, much as modifications were necessary to support dataflow modules for our visualization environment.

Sieve's current design attempts to preserve as much of the original BeanBox functionality as possible, but also addresses the problems described below. Note that these objectives are from the perspective of a developer writing application components for use in Sieve, and not from the perspective of a Sieve user.

4.2.1 Flexible Component Construction

The manner in which objects are added to the workspace must be flexible. In the BeanBox, only a single type of user interaction is available for adding a component to the workspace: The user clicks on the workspace to specify the upper-left corner of the bean's visual representation. The component is then instantiated and added to the workspace when the mouse is released.

This interface mechanism proved problematic when, for example, we attempted to add whiteboard-style tools for annotating dataflow networks. To have the expected behavior, construction of a component such as a line required multiple mouse actions (e.g., clicking and dragging).

As another example, some applications might need to support "drag-and-drop" functionality for component creation. Users might initiate component creation by clicking the mouse on an icon in a palette and dragging onto the workspace. These applications should have the ability to display a visual representation of a new component that follows the user's mouse pointer until the mouse is released.

In addition to issues with the user interface for constructing components, we also encountered the deeper issue of supporting components whose construction was more complex than simple instantiation. BeanBox requires that all beans implement a parameterless constructor. When a bean is added to the workspace, the `instanceOf` method of the `Class` class is used to instantiate the object using its parameterless constructor.

While it is possible to build arbitrarily complex creational mechanisms on top of this simple approach, we wanted to give Sieve-based applications the option of instantiating objects in any way that they choose.

Consider, for example, providing a user of our circuit simulation with a set of fixed-value resistors, rather than a single resistor component whose value could be edited. (Such a configuration might be useful as an exercise for physics students, since constraining the available kinds of resistors would make the exercise more realistic and challenging.) In the BeanBox, each resistor value would require a separate class. Each of these classes would likely be a trivial subclass of a `Resistor` superclass in which the value of the resistor was set to some constant.

We would prefer instead to allow the application developer to simply implement the `Resistor` superclass and set the value of a given instance at construction, perhaps by passing the value as a constructor argument. The application might, for example, determine this value based on what

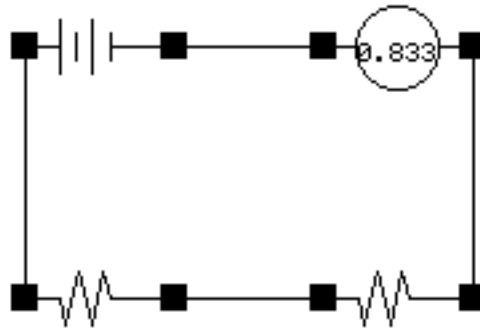


Figure 4.3: Four linked components in a Sieve-based circuit simulation.

the user did to initiate construction. If they dragged the mouse out of the 5 Ohm resistor bin, then the value would be set to 5 Ohms at construction. This approach not only reduces the number of classes needed by the application, but also allows the set of possible values to be configured dynamically.

4.2.2 Flexible Component Linking

To support a wide range of applications, it must be easy for developers to introduce new ways to connect objects on the workspace.

To construct our BeanBox-based prototype of Sieve, it was necessary to introduce a linking mechanism that was completely different from the mechanisms provided by BeanBox. When connecting two dataflow components, this linking mechanism made the necessary method invocations to allow the destination component to retrieve data from the source component. The object that encapsulated this linking behavior was similar to the adapter classes generated by BeanBox to support property binding and event propagation (as illustrated in Figures 4.1 and 4.2). Our new link class, like the BeanBox-generated adapter classes, simply implemented a few lines of code to connect the components in a specific way.

At the user interface level, modifications to the BeanBox were also needed to give these links a distinct appearance. Rather than being invisible, we wanted links between visualization components to appear as a line with an arrow indicating the direction of the dataflow. We also wanted to allow a user of our visualization environment to remove a link by clicking on its representation on the workspace. Adding this behavior required additional modifications.

A different application, however, could easily require both a new kind of linking behavior and a different link appearance. For example, Figure 4.3 shows four connected components in a Sieve-based circuit simulation: a battery, an ammeter, and two resistors. The linking mechanism required to connect components in this application draws links without arrows, and also executes completely different logic from that of the dataflow components in our Sieve-based visualization environment.

Ideally, of course, components should be designed in such a way that they can be connected using the standard JavaBeans linking mechanisms (property binding and event notification). In cases where linking via property binding or event notification is not practical, application developers should be able to implement alternate means for connecting components and for displaying connections on the workspace.

4.2.3 Flexible Component Selection

As in vector-based drawing systems, a component-based environment requires some mechanism for selecting components for editing, moving, or deleting. To support appropriate behavior for different types of components, application developers should be able to define how component selection is accomplished.

In the version of BeanBox that we modified, components were selected by clicking on a thin border surrounding each component. This is sufficient for many types of objects and may, in fact, be the only practical way to implement selection of objects that are themselves designed to handle mouse events. Consider, for example, an applet that supports bitmap drawing. If this applet is added to the BeanBox, then the behavior of the applet may be adversely affected if mouse events generated within the applet are intercepted by the BeanBox. (This reflects a limitation of Java's AWT, which provides no simple way of preventing a visible component from receiving mouse events.) Allowing selection by clicking anywhere within the applet could also simply be annoying, since it would (perhaps inadvertently) de-select any other selected component.

Even though support for adding arbitrary objects to the workspace limits the options available for selection, application developers should be able to implement alternate selection policies if desired. This is particularly true of application components written specifically for use in our collaborative workspace. For example, it might make sense to allow selection of whiteboard objects by clicking on any painted area of the object.

An application developer might also choose to disallow selection of some or all components under certain circumstances. For example, a collaborative application may need to support a floor control policy in which only the user who currently has control of the session may select components. Alternatively, a collaborative application might prevent two or more collaborators from selecting the same component at the same time.

As with component construction, there is potentially a larger issue behind the user interface problems presented by selection. In the BeanBox, selection results in the display of a property editor for the selected component. While this is appropriate for many of the applications for which we envision that Sieve could be used, application developers should have the ability to substitute other behaviors.

4.2.4 Flexible Component Display

Our BeanBox modifications addressed one type of alternate display of components: displaying editors for "invisible" objects. This approach may, however, not be appropriate for all applications.

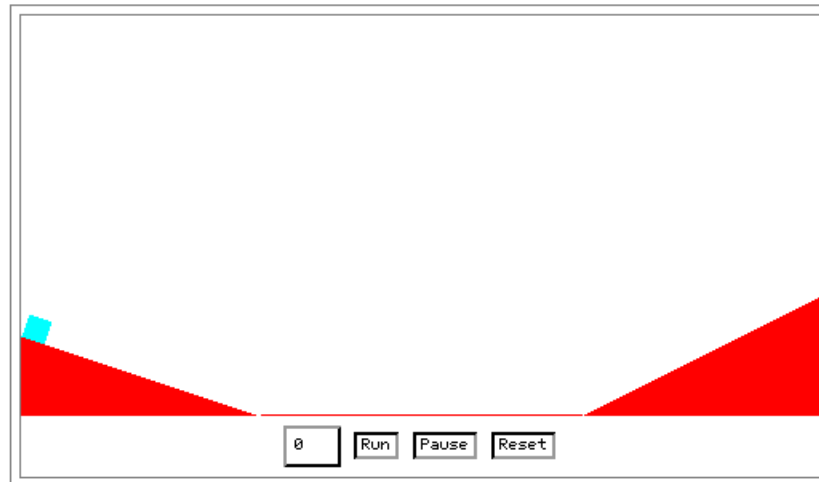


Figure 4.4: A simple simulation component with several sub-components.

In general, an application developer should be able to decide what is displayed on the workspace, and also to decide what relation the visual representation of a component has to the component itself.

This goal is quite similar to that of the Model/View/Controller (MVC) framework [19] for building user interfaces in Smalltalk. In this framework, the Model is the application's invisible representation of an interesting entity. The View is that entity's screen representation, and the Controller defines the response to user input. To allow separation of component functionality from component interface, application developers should be able to easily implement an MVC-style design.

Additionally, the BeanBox implementation of rectangular borders around each component may not be appropriate for all applications. For example, users strongly disliked having rectangular borders around early Sieve-based whiteboard components. Application developers should be able to easily implement alternate means for showing when a component is selected or when a link may be started from or terminated to a component.

4.2.5 Support for Aggregate Components

BeanBox does not provide any mechanism for linking or editing aggregate components. Although many forms of object relations may be categorized as aggregation (see [16] for an overview of types of object aggregations), we are primarily interested in containment. In particular, we would like to support linking and configuration of visual components contained within other visual components.

Consider, for example, the block-and-plane simulation component shown in Figure 4.4. This component implements a simple simulation of a block sliding down an inclined plane, across a flat surface, and up another inclined plane. The simulation generates data describing the position, velocity, and acceleration of the block at regular time intervals.

The block-and-plane component is essentially an aggregate of four parts: the block, the left inclined plane, the right inclined plane, and the flat surface between the two planes. The three planes have properties — such as height and coefficient of friction — that control the behavior of the simulation. Similarly, the block also has a set of properties, including size, shape, and mass. Finally, the simulation has other “environment” properties, such as the acceleration of gravity. These properties are independent of any of the four sub-components.

To make the simulation usable in BeanBox, it is necessary to expose the properties of all four sub-components as properties of their container, the simulation component itself. This makes the simulation component unnecessarily large, and introduces numerous opportunities for error when modifying the implementation of the subcomponents. To avoid these problems, we would like to provide the ability for a component to request that a subcomponent contained within it be made available for editing.

Linking proves to be a more serious problem, as there is no simple way (with a BeanBox-based design) to distinguish which subcomponent a link should be initiated from or terminated on. This would be problematic for a circuit simulation application, where a single component will typically have multiple points to which links can be made. Our environment must therefore provide a way for application developers to implement new linking mechanisms to connect subcomponents.

Since no standard presently exists for describing aggregate beans, our objective is to support linking and editing parts of aggregates not by rigidly defining our own standard, but rather by avoiding limits on what may be edited or linked to.¹

4.2.6 Minimizing Application Development Overhead

The ability to substitute new mechanisms for constructing, selecting, and displaying components should not hinder development of applications that do not need functionality beyond that provided by the BeanBox or by our original BeanBox-based prototype of Sieve. The design must give application writers the opportunity to change elements of the system’s behavior, but default implementations must be available for applications that only need BeanBox-like behavior.

If, for example, an application only requires a new way to connect specific kinds of components, it should not be necessary to implement anything other than a new link class. The developer should then be able to “plug-in” this new class to have it be used by other parts of the system.

4.2.7 Decoupling of Collaboration Support

Collaboration support was almost certainly the most radical departure that we made from the original intent of the BeanBox. In our initial BeanBox-based prototype, extensive modifications were necessary to support broadcasting of workspace changes to collaborating sessions. For example, there was originally no way (using either event listeners or subclassing of BeanBox classes)

¹We do not consider *creation* of aggregate components to be part of this objective. Bean creation – in the sense of using the workspace to create a set of classes that can be packaged and imported into other JavaBeans-compliant tools – represents a different and larger problem.

to detect when a connection was established between two components. Support for this required finding the appropriate points in the BeanBox code where component linking was completed and inserting code to send an event to other collaborating sessions.

This coupling of basic workspace functionality with collaboration support was undesirable for a number of reasons. For situations where collaboration is not needed, this coupling would add unneeded code, since the collaboration support code could not easily be separated from other workspace functionality. It would also make it difficult to substitute different implementations of the underlying collaboration infrastructure. For example, both our prototype and our revised Sieve implementation simulate multicast by sending messages separately to each collaborating session. When true multicast becomes practical, it would be desirable to be able to update Sieve to use it without altering any of the classes that define the behavior of the workspace.

4.2.8 Support for Writing Collaboration-Aware Components

Our BeanBox-based prototype supported the use of *collaboration-unaware* components. In other words, components did not have to be designed to include any explicit collaboration support. Instead, any component that exposed its state using the JavaBeans bound properties mechanism could be shared. This approach supports component state replication at the granularity of properties: A property change in one component is reflected in that component's replicas in all collaborating sessions.

The primary advantage of this design is that it allows a wide range of third-party, "off-the-shelf" components to be used collaboratively on the workspace, without modification, provided that they expose their state using bound properties. There may, however, be some components for which interesting state changes cannot easily be modeled as property changes.

In some cases, for example, property changes may not provide sufficiently fine granularity. Consider a component that implements a simple text editor. One approach to sharing such a component would be to expose the entire content of the editor as a property. Any keystroke that changed the content would then cause a property change, and the new content would be propagated to all collaborating sessions. Such an approach might be sufficient for editing small chunks of text, but would clearly not be desirable once the content grew to more than a few words. Aside from efficiency concerns, simple content replication would almost certainly prevent two or more collaborators from being able to edit the text simultaneously. A better solution for implementing a component such as this would be to propagate only a light-weight description of each change. These changes could then be merged into each collaborator's copy of the editor's content.

To allow use of both collaboration-aware and collaboration-unaware components, our collaborative workspace should support simple property change propagation for all components, and also allow collaboration-aware components to provide more complex means for replicating state.

Chapter 5

The Sieve Framework

This chapter describes our re-design of the Sieve framework, based on the objectives established from experience with our BeanBox-based prototype. We use the term “framework” to describe the core pieces of Sieve that are common to all Sieve-based applications.

Figure 5.1 illustrates the static structure and relationships among the set of objects that all Sieve applications use. (An overview of the UML notation used in this diagram is given in Appendix A.)

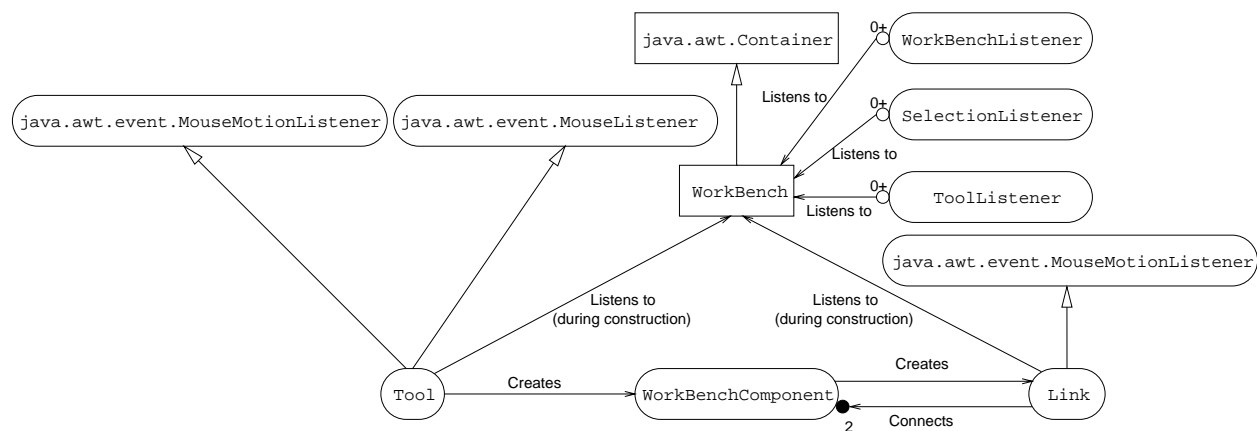


Figure 5.1: High level structure of core Sieve components.

The **Workbench** is the container onto which visual representations of components are placed. These components are accessed through objects which implement the **WorkbenchComponent** interface, and are created by objects which implement the **Tool** interface. **Tool** objects listen to mouse events generated by the **Workbench** to determine how a given component should be created.

Objects which implement the **Link** interface provide mechanisms for connecting components. These also listen for mouse events from the **Workbench**, typically to draw themselves properly as the user in making a connection.

Throughout the design of Sieve, we make extensive use of the delegation event model introduced in the Java 1.1 AWT and JavaBeans. Under this model, events are represented as objects whose class extends `java.util.EventObject`. Event classes typically have names that end with “Event.” For example, events that describe mouse actions are usually instances of `java.awt.MouseEvent`. An event object is propagated from a source object to “listener” objects by invoking a method on the listener and passing the object as a parameter. A listener is an object that implements a sub-interface of the `java.util.EventListener` interface. Listeners have naming conventions similar to those of events. For example, objects that listen for mouse events implement the `java.awt.MouseListener` interface.

Listeners are typically “attached” to sources by invoking `add` and `remove` methods, e.g. `addMouseListener` and `removeMouseListener`. In most cases, an arbitrary number of listeners may be attached to a source object. A more detailed explanation of the delegation event model may be found in [34].

The `Workbench` supports registration of listeners implementing three different interfaces: `WorkbenchListener`, `ToolListener`, and `SelectionListener`. These listeners will be notified when interesting events occur within the `Workbench` or within a component on the `Workbench`. This mechanism is intended to be the primary means by which application developers alter or extend the behavior of the `Workbench`.

Each of the core components shown in Figure 5.1 is examined in greater detail in the following sections. We rely on two standard notations to help describe the design of the Sieve framework, and later to describe the design of specific application components. Classes and interactions between classes are illustrated using the notation of the Unified Modeling Language (UML) [27]. A brief overview of the subset of UML used in this chapter is given in Appendix A. Where applicable, we also use the design pattern terminology of Gamma, et al. [11] to describe compositions of objects. The patterns that we use most commonly are described in Section 2.4.

5.1 The Workbench Class

The `Workbench` class implements a Sieve application’s workspace. It has three primary responsibilities: providing an instance of `java.awt.Container` onto which the visual representations of components may be added, negotiating requests made by links and components, and informing listeners when interesting things happen within the workspace.

The `Workbench` is designed to provide only the minimal services that are likely to be needed by most Sieve-based applications. As such, it enforces very few restrictions on how components may be added, displayed, linked, moved, or removed. As much application-specific functionality as possible is delegated to the other entities such as `Tool`, `WorkbenchComponent`, and `Link` objects.

The `Workbench` is designed to be extended by composition rather than inheritance. For example, it contains no explicit support for component instantiation or collaboration. Instead, these services are provided by interchangeable classes that may be attached to or “plugged into” the `Workbench`.

Workbench
<p>Purpose</p> <p>The Workbench class implements a Sieve-based application's workspace.</p>
<p>Methods</p> <pre> public void setWorkbenchComponentEditor(WorkbenchComponentEditor e) public WorkbenchComponentEditor getWorkbenchComponentEditor() public void setCurrentTool(Tool t) public Tool getCurrentTool() public void setToolRetained(boolean toolIsRetained) public boolean isToolRetained() public void setCurrentSelection(WorkbenchComponent c) public WorkbenchComponent getCurrentSelection() public boolean requestSelection(WorkbenchComponent w) public boolean requestEditor(WorkbenchComponent w) public boolean requestLinkStart(WorkbenchComponent w, Link l) public void componentAdded(WorkbenchComponent wbc, Tool tool) public void componentRemoved(WorkbenchComponent wbc) public void componentBoundsChanged(WorkbenchComponent w) public void componentEdited(WorkbenchComponent w, EditEvent e) public void linkAdded(Link l) public void linkRemoved(Link l) public void addWorkbenchListener(WorkbenchListener l) public void removeWorkbenchListener(WorkbenchListener l) public void addToolListener(ToolListener tl) public void removeToolListener(ToolListener tl) public void addSelectionListener(SelectionListener l) public void removeSelectionListener(SelectionListener l) public void addPropertyChangeListener(PropertyChangeListener l) public void removePropertyChangeListener(PropertyChangeListener l) </pre>

Figure 5.2: The Workbench class.

Figure 5.2 shows the public interface of the `Workbench` class. Following JavaBeans naming patterns, each of the atomic properties of the `Workbench` has `get` and `set` methods. For example, the `currentEditor` attribute may be accessed and modified using instance methods `getCurrentEditor` and `setCurrentEditor`.

Also following JavaBeans patterns, `add` and `remove` methods are provided for attaching and detaching each type of listener. For example, a `WorkbenchListener` may be added by calling `addWorkbenchListener` and removed by calling `removeWorkbenchListener`.

Registered `WorkbenchListener` instances will be notified when components are added, edited, moved, resized, selected, de-selected, and removed. They are also notified when links are started, completed, and removed. `SelectionListener` implementations are responsible for granting a component permission to select itself for movement or editing. This will be discussed in greater detail in Section 5.2.2. `ToolListener` implementations are notified when a new “current tool” is registered with the `Workbench`, and when a tool is used to create a component. To support the use of `Workbench` instances as beans, `PropertyChangeListener`s may also be registered.

The remaining `Workbench` methods may be divided into two general categories: those that notify the `Workbench` of changes to a component or request a service for a component, and those that notify the workbench of changes to a link between components. (These methods will be described in more detail in the following sections.)

5.2 The WorkbenchComponent Interface

Each component appearing on the Sieve workspace corresponds to an object that implements the `WorkbenchComponent` interface. Figure 5.3 shows the methods in the `WorkbenchComponent` interface. The instance of `java.awt.Component` returned by `getView()` is the visual representation of the `WorkbenchComponent` that will appear on the workspace. The underlying object that the component represents is returned by `getModel()`. For components that are inherently visual, these two methods will typically (though not necessarily) return the same thing.

Methods are also provided to determine and specify whether or not the component is selected, and to query and set a unique identifier for the component. This unique identifier is used by Sieve’s collaboration support classes to identify component replicas across multiple collaborating sessions. This will be described in greater detail in Chapter 6.

The `WorkbenchComponent` interface addresses a number of our design objectives. By separating a component’s visual representation from the component itself, the application can control all aspects of a component’s presentation on the workspace: what visual representation is shown, how selection is accomplished, how selection is indicated, how the component may be moved, how subcomponents are selected or otherwise manipulated, and how links may be started and terminated.

WorkbenchComponent
<p>Purpose</p> <p>Every component in the Sieve workspace is represented by an implementation of the <code>WorkbenchComponent</code> interface.</p>
<p>Methods</p> <pre> public Object getModel() public Component getView() public boolean isSelected() public void setSelected(boolean selected) public void remove() public UniqueID getUniqueID() public void setUniqueID(UniqueID id) </pre>

Figure 5.3: The `WorkbenchComponent` Interface.

5.2.1 Types of `WorkbenchComponent` implementations

`WorkbenchComponent` implementations may be divided into two broad categories: “wrappers” and Sieve-specific components.

Wrapper implementations are used as containers for arbitrary objects, allowing non-Sieve-specific components to be used on the workspace. Sieve’s default implementation of the `WorkbenchComponent` interface can contain any Java object and allow it to be manipulated in Sieve much as it would be manipulated in `BeanBox`. For example, the dataflow components used in the visualization environment described in Section 3.2 were not designed specifically for use in Sieve. The functionality needed to make the components manipulable on the Sieve workspace is achieved by dynamically composing a filter or visualization component with an instance of the default wrapper class. Wrapper `WorkbenchComponent` implementations can also be written for objects that implement a particular Java interface or extend a particular class.

If components are being specifically developed for use in Sieve, the components themselves can implement the `WorkbenchComponent` interface. The circuit simulation components described in Section 3.3 were designed this way. Classes representing resistors, batteries, and other circuit components are all subclasses of a single `CircuitComponent` class that implements the `WorkbenchComponent` interface.

Each approach has advantages and disadvantages. A single wrapper implementation can be used with numerous component classes. The component classes need not have any knowledge that they are to be used in Sieve. However, the more general the wrapper, the less functionality it is able to provide. Our default wrapper, for example, cannot assume the presence of any behavior other than that provided by `java.lang.Object`. A wrapper for subclasses of `java.applet.Applet`, on

the other hand, could make additional assumptions about available behavior and methods. Such a wrapper could, for instance, provide buttons for starting and stopping the applet.

Having a component implement the `WorkbenchComponent` interface directly gives the developer complete control over how the component is presented, as well as how the user is allowed to interact with the component. As we will discuss later in this chapter, Sieve-specific components may implement special linking, editing, and selection mechanisms. The disadvantage is that such components are less likely to be usable outside of the Sieve environment, and they generally require more effort to produce.

5.2.2 Component Selection

Component selection in Sieve serves much the same purpose as selection in object-based drawing applications. The selected component can typically be edited, moved, resized, or deleted.

The selected component will usually be drawn differently (e.g., with handles or a solid border) and will generally be editable through an editor panel. Note that to preserve the ability to use arbitrary components in Sieve, components whose visual representations allow manipulation will be active at all times, regardless of whether they are selected or not. In other words, if a given component's `getView` method returns a button object (or a panel containing button objects), Sieve does not attempt to disable the buttons when the component is not selected. Although this would be possible technically, it would likely interfere with the component's behavior. The state of an object such as a button is essentially part of the internal state of the component that contains the button. By disabling the button, Sieve would be altering the internal state of the component in a way that the component's designer almost certainly did not intend or consider.

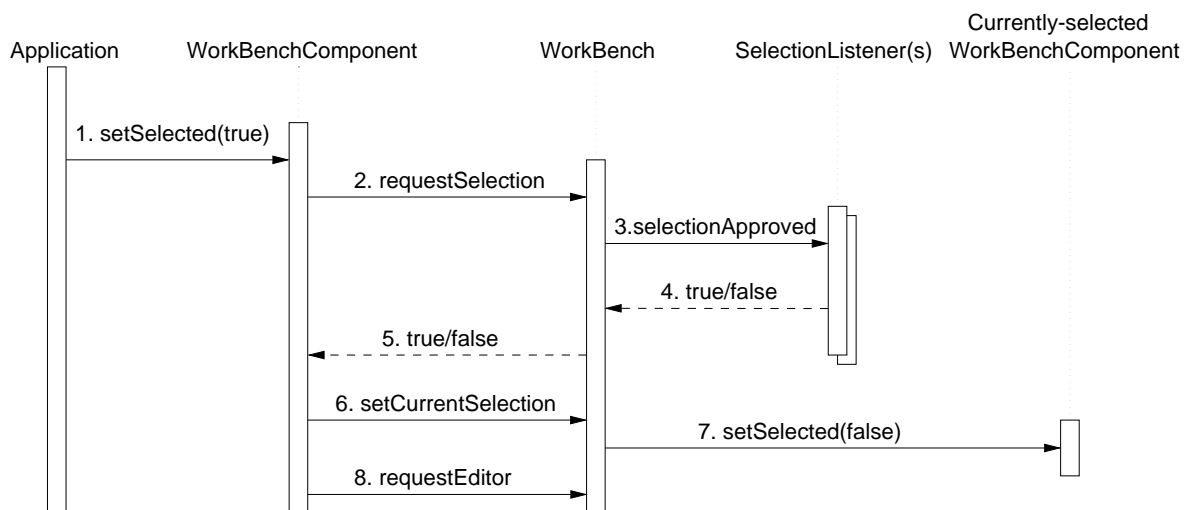


Figure 5.4: Interaction diagram showing `WorkbenchComponent` selection.

Figure 5.4 shows the interactions between objects in a typical component selection scenario. In this scenario:

SelectionListener
<p>Purpose</p> <p>The <code>SelectionListener</code> interface allows applications to add new selection approval policies.</p>
<p>Methods</p> <pre>public boolean selectionApproved(WorkbenchComponent c)</pre>

Figure 5.5: The `SelectionListener` Interface.

1. Component selection is initiated by some piece of application code. Often this will be the `WorkbenchComponent` itself, when, for example, a mouse click is detected on the component's border. Regardless of who initiates selection, the `setSelected` method of the `WorkbenchComponent` instance should be called. (As with any other property, the `WorkbenchComponent` should verify that the call to `setSelected` would indeed change its selected state. In other words, duplicate requests to change a property should always be ignored.)
2. Well-behaved components should then request permission from the `Workbench` to become selected by calling the `Workbench` instance's `requestSelection` method.
3. Upon receiving a request for selection, the `Workbench` will ask all registered `SelectionListener` objects (Figure 5.5) if the `WorkbenchComponent` may proceed with selection. Any `SelectionListener` may veto the selection request by returning `false` from its `selectionApproved` method. In a collaborative application, for example, this mechanism could be used to ensure that a given component can only be selected by only one user at a time.
4. If selection may proceed, the `Workbench` will return `true` from the `requestSelection` invocation.
5. Upon receiving permission to proceed, the `WorkbenchComponent` should call the `Workbench` instance's `setCurrentSelection` method to inform the `Workbench` that it is now the current selection. The `WorkbenchComponent` would typically also change its appearance at this point to indicate that it was selected. (This is also the point at which any registered `WorkbenchListener` objects will be notified that a new component has been selected.)
6. If another `WorkbenchComponent` instance was already selected, the `Workbench` will set that component's selected state to `false`.
7. If the `WorkbenchComponent` wants a property editor presented to the user, it may request one by invoking the `Workbench` instance's `requestEditor` method.

This approach allows a great deal of flexibility. New application-wide selection policies may be introduced by simply attaching new `SelectionListener` objects. Components are given complete

freedom over the way in which a user interacts to achieve selection, as well as the way in which selection is indicated. Finally, components are given the opportunity to use the standard `BeanBox`-style editor, but are not required to do so.

5.2.3 Component Editing

Components in the Sieve workspace are generally edited by changing their exposed properties. These changes may be effected in several different ways. In the scenario above, the `WorkbenchComponent` instance requests that the `Workbench`'s property editor be presented to the user to allow editing. The `WorkbenchComponent` may, however, also choose to display a property editor or customizer as the component's visual representation. As discussed previously, this is particularly useful for otherwise "invisible" beans.

The `WorkbenchListener` interface provides support for notifying listeners when a component has been edited. One approach to implementing this could be to have the `Workbench` register itself as a `PropertyChangeListener` to every component that is added to the workspace. Such an implementation has several drawbacks, most notably that it may not be able to detect changes made to unbound properties via a property panel or other editor. Another possible problem with this approach is while (ideally) all edits result in property changes, not all property changes directly reflect a user edit.

Our solution to this problem is to allow any knowledgeable entity to notify the `Workbench` of a change by calling the `Workbench` object's `componentEdited` method. This allows any object that implements an editor to notify the `Workbench` of a change, regardless of whether that change was to a bound property or not.

`WorkbenchComponent` implementations may also notify the `Workbench` of changes. Our default wrapper `WorkbenchComponent`, for example, simply registers itself as a property change listener to the arbitrary object that it wraps, and forwards any change notifications to the `Workbench`. Application-specific `WorkbenchComponent` implementations may also be able to filter property changes, ignoring those changes that are not actually the result of a user editing the component. In our Sieve-based circuit simulation, for example, a user can change the resistance of a resistor component. If the component is in an active circuit, then this will also change the current passing through the resistor. Of the two changes, only the first change represents an edit.

Since this model allows multiple objects to detect and report property changes to the `Workbench`, the `Workbench` is responsible for ignoring duplicate notifications.

Changes to a component are described by an instance of a subclass of the `EditEvent` abstract class, shown in Figure 5.6. An `EditEvent` instance's `applyChange` method will apply the change described by the event to a target object. The `isObsoletedBy` method provides a generic way to determine if one change overrides, or obsoletes, another.

As noted previously, most edit operations on Sieve components are described by property changes. Such changes are typically detected when the component fires a `PropertyChangeEvent`. To handle edits that result in property changes, we have implemented the `PropertyEditEvent` subclass of `EditEvent`. The `PropertyEditEvent` class provides a wrapper around a `PropertyChangeEvent`.

EditEvent	
Purpose	EditEvent instances describe changes to a WorkbenchComponent.
Methods	<pre>public void applyChange(Object target) public boolean isObsoletedBy(EditEvent newEvent)</pre>

Figure 5.6: The EditEvent Abstract Class.

The `applyChange` method simply locates the appropriate property setter method in the target object and invokes it with the property’s new value. The `isObsoletedBy` method will return true if the specified `EditEvent` is a `PropertyEditEvent` and the property names are equal.

The `EditEvent` is an example of the Command design pattern [11]. It encapsulates an operation such that another object can apply the operation without knowing exactly what it does. To apply the change represented by an `EditEvent` to another object, it is only necessary to ensure that the object passed to the `applyChange` method is of the appropriate type. All other details are hidden.

5.2.4 Component Movement, Reshaping, and Removal

Sieve includes explicit support for tracking the bounds (size and position) of a `WorkbenchComponent` instance’s visual representation.

When a component is moved or reshaped, the `WorkbenchComponent` is responsible for calling the `Workbench` object’s `componentBoundsChanged` method. The `Workbench` will then get the component’s new bounds by getting the bounds of the `java.awt.Component` instance returned by the `WorkbenchComponent`’s `getView` method. The `Workbench` then notifies any `WorkbenchListeners` of the change.

As the visual representation of a component may be distinct from the underlying object, each `WorkbenchComponent` class must implement a `remove` method that removes the visual representation and performs any cleanup of the underlying object (the component’s “model”). The `Workbench`’s `componentRemoved` method should be called once the `WorkbenchComponent` has completed removal and cleanup of the component.

5.3 The Tool Interface

The `Workbench` never instantiates a `WorkbenchComponent` directly. Instead, an object that implements the `Tool` interface (Figure 5.7) is registered with the `Workbench` as the “current tool.” While

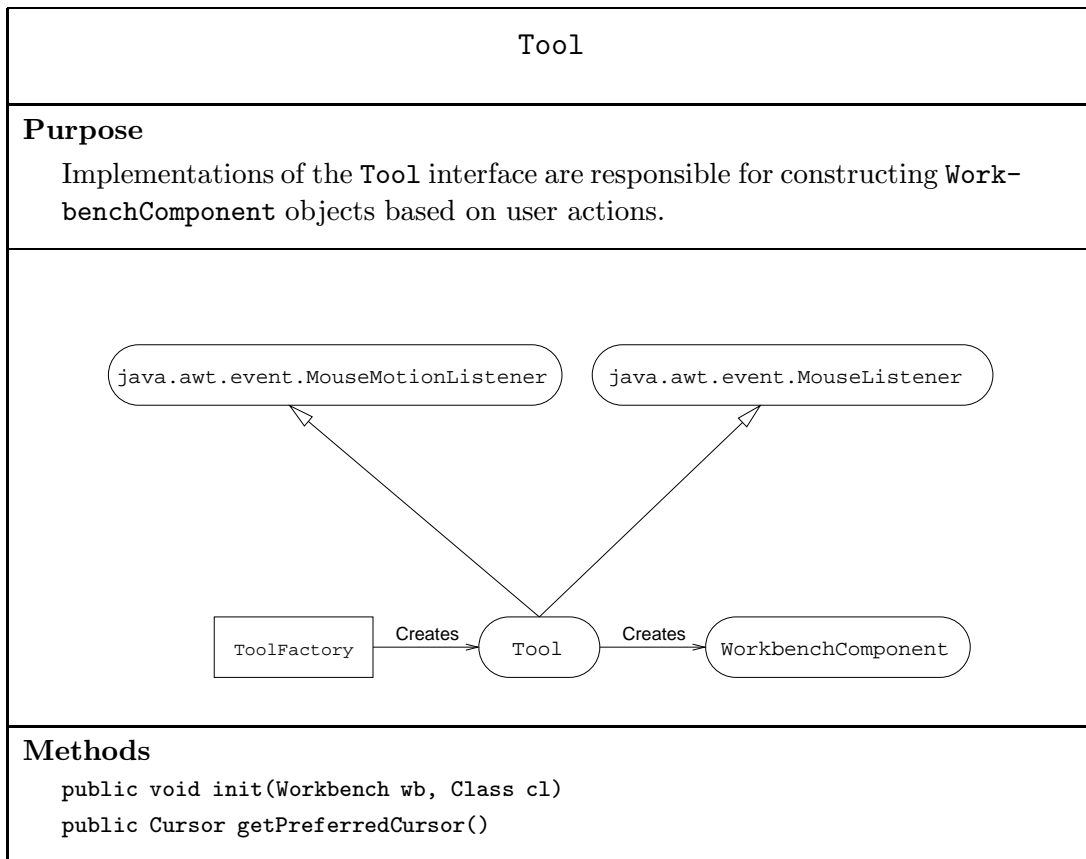


Figure 5.7: The Tool Interface.

registered in this way, a `Tool` will receive all mouse events detected by the `Workbench`. It can then use information about the user’s actions to construct a new component in any way that it chooses.

`Tool` objects are responsible for informing the `Workbench` when a new component has been created by calling the `Workbench`’s `componentAdded` method. An alternative would have been to have the `Workbench` listen for new instances of `java.awt.Component` to be added. However, forcing the `Tool` to explicitly inform the `Workbench` of component addition allows a `Tool` to temporarily add visual elements to the workspace during component construction without committing to the addition of a new `WorkbenchComponent`. Consider a `Tool` that implements “drag-and-drop” behavior when adding components. It should add just the visual representation of the component to the workspace until the user commits to the component addition. At that time, the `Tool` will inform the `Workbench` that a new component has been added.

Figure 5.7 shows the class hierarchy of the `Tool` interface. To receive all types of mouse events, the interface extends both the `MouseListener` and `MouseMotionListener` interfaces. This results in the interface containing a rather large number of methods (seven from the mouse interfaces and two more that are specific to the `Tool` interface). Keeping with AWT convention, a no-op

ToolFactory	
Purpose	The ToolFactory class provides a mechanism for mapping classes to the Tool implementations that can be used to create them.
Methods	<pre> public static ToolFactory getToolFactory() public void registerTool(String componentName, Class toolClass) public Tool getTool(Class componentClass, Workbench wb) </pre>

Figure 5.8: The ToolFactory class.

implementation of the Tool interface – the SimpleTool class – is provided. This class may be subclassed, with only the relevant methods overridden.

Since constructors may not be specified in a Java interface, the interface contains an `init` method that takes as parameters the current `Workbench` and a `Class` object representing the kind of component that is to be added. This allows a single `Tool` implementation to be written in such a way that it can construct an arbitrary number of different kinds of components. The `getPreferredCursor` method allows a `Tool` to specify a cursor to be used by the `Workbench` while the `Tool` is active. The `Tool` could also simply set the cursor on the `Workbench`, but having the `Tool` specify a cursor allows an application to use the preferred cursor in other ways. For example, it might make sense to use the preferred cursor on a palette component from which tools are selected.

`Tool` objects are often used once and then discarded, but this need not be the case. When it is desirable to allow the user to use the same tool repeatedly, the `Workbench` contains methods `isToolRetained` and `setToolRetained`. The `toolRetained` property allows the current tool to be kept as the current tool after it has been used. If this property is set to false, then the current tool will be cleared after a component has been added. Our design makes this a property of the `Workbench` (and not of each `Tool`) in the belief that the application will be able to decide when this behavior is appropriate more often than a given `Tool` implementation. Placing the property in the `Workbench` allows the application to set the behavior in a single place rather than in each `Tool` instance.

While applications may obtain instances of `Tool` objects in any way they choose, Sieve provides a standard mechanism for locating an appropriate tool based on the class of the `Workbench-Component` implementation that is needed. The `ToolFactory` class implements a simple registry that associates `Tool` implementations with component classes. An application can then use the `ToolFactory` to produce an instance of a `Tool` object that knows how to generate a `Workbench-Component` representing any Java class. If no specific `Tool` implementation has been registered for a particular class, the `ToolFactory` searches for a `Tool` implementation for any ancestor of the class. The `ToolFactory` first looks for an explicitly registered `Tool` class. If no such class has been

registered, then it appends the string “Tool” to the class name and checks for the existence of that class.

If no specific implementation can be found, a default `Tool` is provided that implements `BeanBox`-like behavior. This `Tool` implementation listens for a `mouseClicked` event, instantiates an instance of the component class, then wraps the component in an instance of the default `WorkbenchComponent` implementation. The `Tool` then obtains a reference to the component’s visual representation by calling the `WorkbenchComponent` instance’s `getView` method. The visual representation is added to the workspace at the location of the mouse click. In this way, the `ToolFactory` and the default `Tool` implementation partially address our design objective of providing `BeanBox`-like defaults for all changeable components of the system.

More importantly, the design of the `Tool` interface and `ToolFactory` class address our design goal of supporting flexible component construction. Neither the `Workbench` nor individual component implementations need to have knowledge of how components are created. To add a new type of component creation, a developer can simply create a new implementation of the `Tool` interface and make it available to the `ToolFactory` either by explicitly registering it or by naming it appropriately. Since a single `Tool` implementation can be registered for an entire hierarchy of classes, one `Tool` can implement the logic for creating many different types of components. Conversely, the way in which a given type of component is created can be changed by registering a new `Tool` implementation for it.

The `ToolFactory` class is an example of both the Singleton and Abstract Factory design patterns [11]. `Tool` implementations may be considered instances of the Builder creational design pattern, as they separate the construction of potentially complex objects from the representation of those objects. Our default `Tool` implementation, for instance, implements sufficient logic to construct and add to the workspace any Java object that has a parameterless constructor.

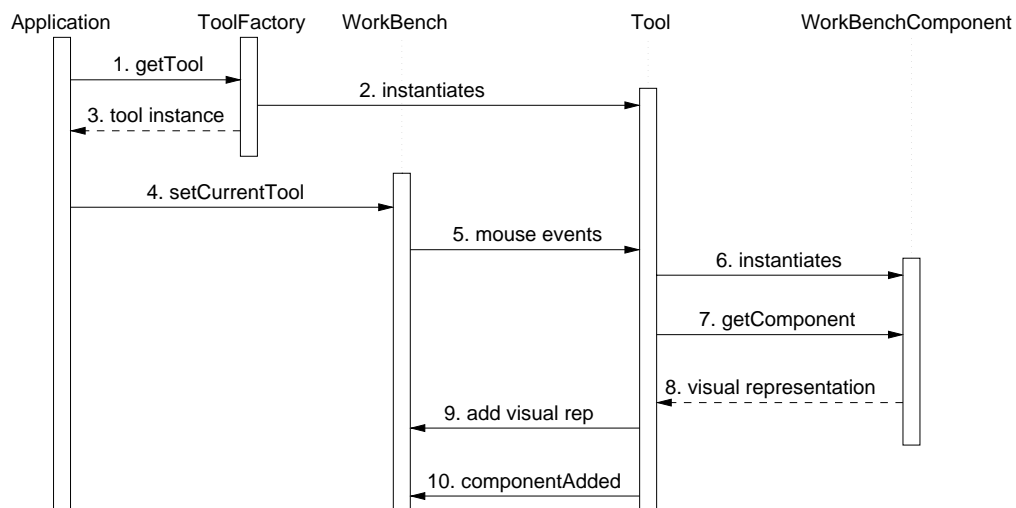


Figure 5.9: Interaction diagram showing `WorkbenchComponent` creation by a `Tool` instance.

Figure 5.9 illustrates the typical process by which a `Tool` object is obtained and used to create a `WorkbenchComponent`. In this scenario:

1. Application code requests an instance of a `Tool` object for a specific kind of component from the `ToolFactory`. This will often be done by some sort of toolbar or palette of tools. (The application may, of course, obtain a `Tool` object in other ways, including direct instantiation. Section 5.6.2 discusses this in greater detail.)
2. The `ToolFactory` determines the type of `Tool` that is appropriate for instantiating the specified type of component.
3. The `ToolFactory` returns a `Tool` instance.
4. The application code tells the `Workbench` to use the new tool as its “current tool.”
5. The `Workbench` sends all mouse events to the `Tool`.
6. The `Tool` object instantiates a new `WorkbenchComponent` object in whatever way is appropriate.
7. The `Tool` object typically asks the new component for its visual representation by calling the `WorkbenchComponent` method `getView`.
8. At some point, the visual representation of the component is added to the `Workbench`.
9. At some later point, the `Tool` informs the `Workbench` that the new component has been added. Following notification that a component has been added, the `Workbench` may set its current tool to `null`.

The mechanism allows `Tool` instances to use as much or as little information about mouse activity as needed to determine how to instantiate a component. To produce `BeanBox`-like behavior, a tool need only listen for the mouse to be clicked. The new component can then be instantiated and added at the point where the click occurred. For example, a tool for creating whiteboard-style components can listen for all of the points that the mouse is dragged over and generate a curve or polygon through those points. For drag-and-drop behavior, a tool might also listen for points over which the mouse is dragged, and move the visual representation of the components to each of those points. Such a tool would then “drop” the component (and notify the `Workbench`) when mouse release is detected.

5.4 The Link Interface

Links are objects that are responsible for encapsulating the logic required to connect components. They may represent connections in which the connected components communicate with each other directly. They may also simply indicate associations of which the connected components are themselves unaware. Links are also responsible for producing an appropriate visual representation of the connection on the workspace. The `Link` interface is shown in Figure 5.10.

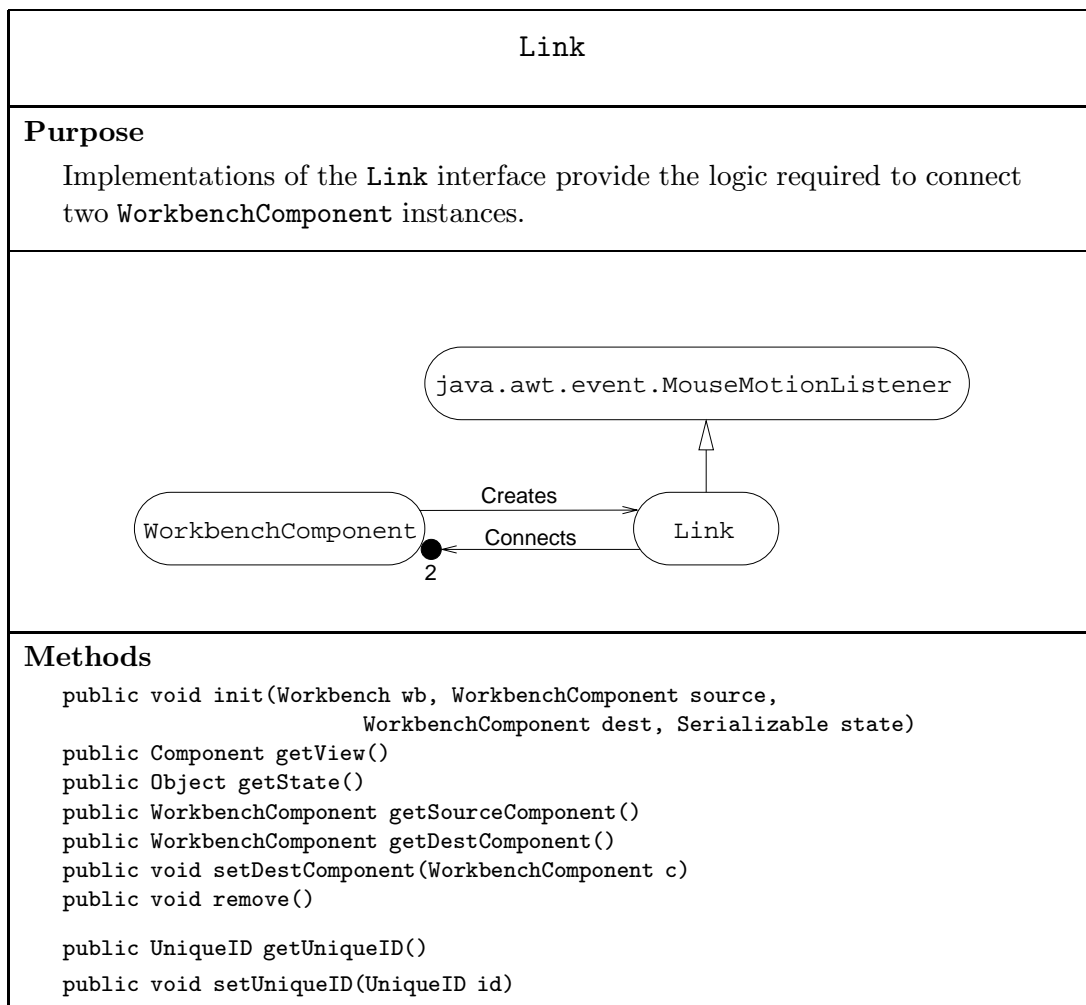


Figure 5.10: The Link Interface.

As with the `WorkbenchComponent` interface, our design does not require that `Link` implementations actually be instances of `java.awt.Component`. Instead, the `getView()` method returns a reference to the object responsible for drawing the link. This allows separation of the link “model” from the “view” of the link presented to the user. `Link` implementations can encapsulate the logic behind the connection separately from the visual representation, and multiple `Link` implementations can share code for drawing common types of links.

Creating a Link

New connections between components are typically initiated by a `WorkbenchComponent` object. For example, a `WorkbenchComponent` might attempt to start a link when the user right-clicks on the border around a component. Prior to starting a link, the `WorkbenchComponent` must first request permission from the `Workbench` by calling `requestLinkStart`. The `Workbench` may choose to deny

the request if, for example, another link has been started by a different `WorkbenchComponent`. This precaution is necessary because Sieve places no restrictions on what user interface mechanism a given `WorkbenchComponent` provides to allow a user to start a link.

If the request is granted, the new `Link` object will be added as a `MouseMotionListener` to the `Workbench`. This allows the `Link` to update its visual representation as the user's mouse pointer is moving to the destination component.

Link termination is somewhat more difficult. To allow the potential for linking arbitrary components, it is not desirable for a given `WorkbenchComponent` implementation to determine when it may be linked to. To solve this problem, we allow any request for component selection to be interpreted as an opportunity for terminating a link. If the `Workbench` has granted a request to start a link and a `WorkbenchComponent` requests selection, the `Link` object's `setDestComponent` method is invoked with a reference to the component that has requested selection.

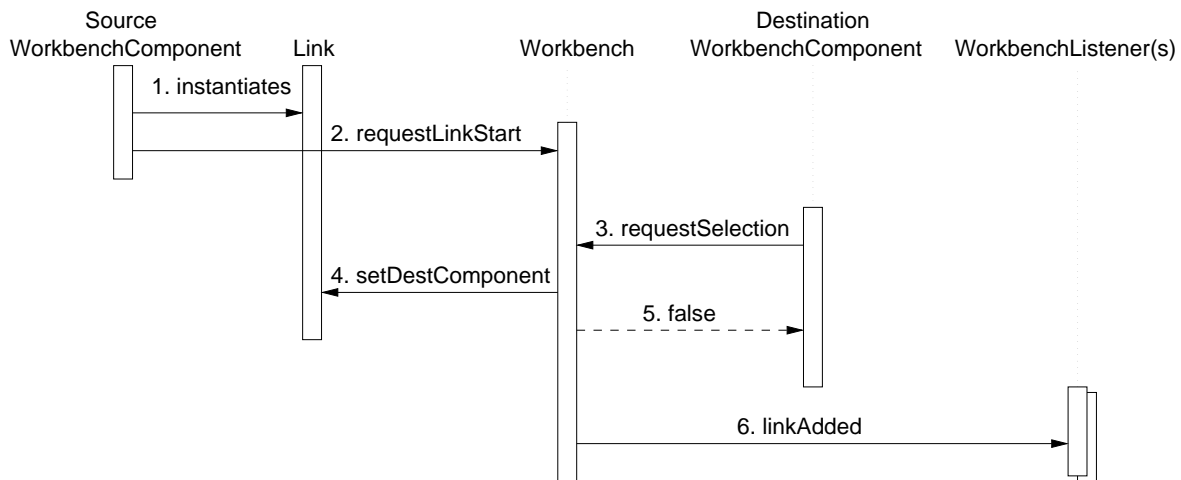


Figure 5.11: Interaction diagram showing simple `Link` creation.

Figure 5.11 illustrates the process by which a simple link between two components is established. In this scenario:

1. The source component first creates a link object. Our default BeanBox-style `WorkbenchComponent` implementation, for example, tries to create a link whenever the user right-clicks on the border of a component.
2. The source component requests permission from the `Workbench` to start the link. The `Workbench` will approve the request if there is not already an unterminated link.
3. At some later time, another component requests permission from the `Workbench` to become the current selection. In our default `WorkbenchComponent` implementation, this will happen any time the user left-clicks on the component's border.

4. The **Workbench** will offer the destination component to the “current” link object. The **Link** object may inspect the **WorkbenchComponent** that has requested selection and throw an exception if it is incompatible with the link.
5. Regardless of whether the **Link** accepts the **WorkbenchComponent** that requested selection, the destination component’s selection request is denied.
6. The **Link** object may throw an exception if it is unable to connect the two components for any reason. If the **Link** does not throw an exception, then all registered **WorkbenchListener** objects are notified that the link has been added.

To support canceling an unterminated **Link**, the **Workbench** listens for a mouse click on the workspace itself. If the user right- or left-clicks on the workspace and a **Link** is being created, the **remove** method of the **Link** object is called.

Removing a Link

Link objects may be removed by invoking their **remove** method. The implementation of this method should both remove the visual representation of the link and execute any necessary logic to disconnect the source and destination components.

Each **Link** implementation is free to define what user interface mechanism (if any) is used to remove themselves. For example, some implementations may choose to remove themselves whenever a mouse click is detected. Others may prompt the user for confirmation, present a pop-up menu of options, or check some global “mode” setting before calling their **remove** methods.

Supporting Complex Connections

From the perspective of the **Workbench**, links connect **WorkbenchComponent** objects. In the context of a specific application, however, links may actually be connecting sub-components of an aggregate component. For example, links connecting **WorkbenchComponent** objects in an electric circuit simulation typically connect one of several “terminals” on a component to a terminal on another component.

This does not present a significant problem for a stand-alone Sieve-based application, since **Link** implementations are able to negotiate with their source and destination components to determine exactly how linking is to be accomplished. There is often, however, no way for an object external to the three parties involved in this negotiation (the **Link** and two **WorkbenchComponents**) to know the details of how the connection was actually accomplished. This is problematic if the link needs to be re-created at a different time and/or place, where reconstructing the exact events that led to link creation may not be possible. When Sieve is used collaboratively, for example, each link will need to be constructed in each replica of the workspace.

Our solution to this is to introduce an optional “state” attribute in our definition of the **Link** interface. Each **Link** implementation that has state information other than its two endpoints should return an object from the **getState** method. When this object is passed to a new **Link**

object's `init` method (along with a reference to a `Workbench`, and source and destination `WorkbenchComponent` objects) it should allow the new `Link` to construct itself in such a way that it is an exact replica of the `Link` on which `getState` was called.

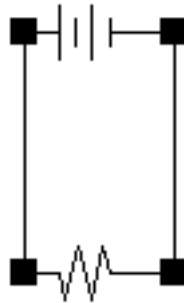


Figure 5.12: Two linked circuit components in a Sieve-based circuit simulation.

For example, consider a link between the terminals of the components in the electric circuit simulation shown in Figure 5.12. In this scenario, the battery and the resistor are linked to each other twice. Each of the two links may therefore have the same source and destination components, but each is linked to a different pair of subcomponents (component “terminals” in this case). The state data for each of these links would simply contain enough information to determine which terminals of the source and destination components are connected.

This solution is an example of the Memento design pattern [11]. It preserves encapsulation while still allowing external objects to obtain a representation of the object's internal state. The state representation is opaque to the external object. It may store the state object, but cannot manipulate it in any way. This approach also allows each `Link` implementation to determine how much or how little information needs to be stored, and therefore places little or no burden on simple `Link` implementations.

Finding Appropriate Link Implementations

Application-specific `WorkbenchComponent` implementations will often know what `Link` implementations are appropriate at the time that a user starts to make a link. For applications that support linking of arbitrary objects, however, it may not be possible to determine what `Link` implementation is appropriate until the link has been terminated. For these instances, Sieve includes two classes that support locating and constructing appropriate `Link` implementations at run-time.

The `LinkFactory` class (Figure 5.13) provides methods for obtaining a list of compatible `Link` classes for connecting a given source component to a given destination component. The `LinkFactory` is similar to the `ToolFactory` in that it is an example of an Abstract Factory and is implemented as a Singleton object. The `getLinkFactory` method provides an interface for clients to obtain a reference to the `LinkFactory` object.

LinkFactory
<p>Purpose</p> <p>The <code>LinkFactory</code> class is used to find <code>Link</code> implementations for connecting <code>WorkbenchComponent</code> instances.</p>
<p>Methods</p> <pre> public static LinkFactory getLinkFactory() public void registerLink(Class sourceClass, Class destClass, Class linkClass) public Class[] getLinkTypes(WorkbenchComponent srcComponent, WorkbenchComponent destComponent) public Class[] getLinkTypes(Class srcClass, Class destClass) public void clearCache() </pre>

Figure 5.13: The `LinkFactory` class.

Like the `ToolFactory` class, the `LinkFactory` supports both explicit registration of `Link` classes as well as dynamic lookup by class name. A particular `Link` implementation may be explicitly registered by calling the `LinkFactory`'s `registerLink` method, passing source and destination classes, along with the `Link` class to be used for connecting them. For example, consider the following invocation of `registerLink`:

```
LinkFactory.getLinkFactory().registerLink(AppletController.class,
    Applet.class, AppletLink.class);
```

After this call had been made, an instance of the `AppletLink` class would be used when the user attempts to connect an `AppletController` instance to an `Applet` instance.

If no `Link` implementation has been explicitly registered for a given source and destination pair, then the `LinkFactory` attempts to find a `Link` implementation based on the class names of the source and destination. The algorithm used by the `LinkFactory` to find a list of compatible links is as follows:

1. Generate an ordered list `sourceClasses` of all super classes and interfaces of and interfaces of the source class. This list is built as follows:
 - (a) Append the current class
 - (b) Append all interfaces implemented by the current class
 - (c) If the current class is not `java.lang.Object`, repeat with the current class's superclass.

For example, consider a class `MyObject` that extends the Java class `java.lang.Object` and implements the interface `MyInterface`. The `sourceClasses` list would contain `<MyObject, MyInterface, java.lang.Object>`.

2. Generate an ordered list `destClasses` of all the destination class's superclasses and interfaces, using the same process used to generate the `sourceClasses` list.
3. Initialize an empty list of compatible links
4. For each class `sc` in `sourceClasses`:
 - (a) For each class `dc` in `destClasses`:
 - i. Check for a `Link` class explicitly registered for connections between `sc` and `dc`. If such a class is registered, add it to the end of the list of compatible links.
 - ii. If `sc` is not a "system class" (if its package name does not begin with "java."), check for a `Link` class whose name is the fully-qualified name of `sc` with the class name of `dc` and the string "Link" appended to it. If such a class exists and implements the `Link` interface, add it to the end of the list of compatible links. For example, if `sc` is the class `visualizations.BarChart` and `dc` is the class `java.awt.Component`, the `LinkFactory` will look for a class named `visualizations.BarChartComponentLink`.
5. Return the list of compatible links

Depending on the depth of class hierarchies, this process can become expensive. Checking for a large number of explicitly registered classes is, in general, fast. Checking for a large number of potentially non-existent classes may be slow, particularly if – as may be the case of a Sieve session running inside an applet – each such check requires requesting a class file from a web server. To make lookups faster in this scenario, `LinkFactory` uses a simple caching mechanism that saves the list of compatible link classes for each source and destination. This makes Sieve somewhat less dynamic, since the cache must be flushed any time new classes are made available. There is no way to detect when new classes appear, so the cache must be cleared by explicitly calling the `clearCache` method.

It may be the case that multiple `Link` implementations will exist for a given source and destination. The `LinkFactory` therefore returns an array of all compatible classes, with the most "specific" implementations appearing first in the list. For example, a `Link` implementation for connecting a `java.awt.Applet` to a `java.awt.Button` would appear before an implementation for connecting a `java.awt.Object` to `java.awt.Object`.

When our default `WorkbenchComponent` implementation initiates a link, the `Link` object that is first created is an instance of the `LinkProxy` class. When the link is terminated on a destination component, the `LinkProxy` object uses the `LinkFactory` to find all compatible link types between the source and destination components. Depending on configuration, the `LinkFactory` will then either choose the first (and therefore most specific) `Link` class, or prompt the user to choose a `Link` class from a list of compatible `Link` classes. An instance of this class will then be created and will essentially replace the `LinkProxy` object.

The `LinkFactory` and `LinkProxy` classes give application designers considerable flexibility in the way that linking functionality is implemented. Applications that only need to connect application-specific `WorkbenchComponent` instances may have those instances construct an application-specific

Link object. Applications that require only BeanBox-like functionality can use the default **WorkbenchComponent** implementation and simply register application-specific **Link** implementations with the **LinkFactory**.

Convenience Classes for Link Implementations

Many application-specific link implementations will have similar functionality. Often, these will differ only in the small segments of code that actually connect and disconnect the source and destination components. To simplify implementing such **Link** classes, Sieve provides a **SimpleLink** class with generic linking functionality, e.g., drawing an arrow between the two components. Subclasses may simply provide application-specific linking functionality by implementing **link** and **unlink** methods.

For cases where a subclass of **SimpleLink** is not appropriate, Sieve also includes a **SimpleLinkComponent** class that implements just the visual representation of a link.

5.5 The WorkbenchListener Interface

External objects can be notified when interesting things happen on a **Workbench** instance by registering a **WorkbenchListener** object using the **Workbench** class's **addWorkbenchListener** method. As shown in Figure 5.14, objects that implement the **WorkbenchListener** interface will be notified when components are added, removed, selected, de-selected, resized, moved, or edited. They will also be notified when links are added or removed.

WorkbenchListener	
Purpose	Registered WorkbenchListener objects are notified of changes to the Workbench .
Methods	<pre> public void componentAdded(WorkbenchComponentEvent e) public void componentRemoved(WorkbenchComponentEvent e) public void componentSelected(WorkbenchComponentEvent e) public void componentDeselected(WorkbenchComponentEvent e) public void componentBoundsChanged(WorkbenchComponentEvent e) public void componentEdited(WorkbenchComponentEvent e) public void linkAdded(LinkEvent e) public void linkRemoved(LinkEvent e) </pre>

Figure 5.14: The **WorkbenchListener** interface.

WorkbenchComponentEditor
<p>Purpose</p> <p>The <code>WorkbenchComponentEditor</code> interface defines the minimal behavior of an editor for components in the Sieve workspace.</p>
<p>Methods</p> <pre>public void setTarget(WorkbenchComponent c)</pre>

Figure 5.15: The `WorkbenchComponentEditor` interface.

Keeping with AWT conventions, the `WorkbenchListenerAdapter` class provides a no-op implementation of this interface. Subclasses of `WorkbenchListenerAdapter` can then override only the interface methods that they actually need.

The radar view (described in Section 3.1) provides an example of how this interface may be used. To stay synchronized with the workspace content, the radar view must be told to update itself whenever anything changes on the `Workbench`. The necessary method invocations could have been included in the `Workbench`, but this would couple the `Workbench` to the radar view implementation. To avoid this, a simple `WorkbenchListener` object is registered with the `Workbench`. When any event is received, this object invokes a method on the radar view to force an update.

5.6 Other Sieve Components

In addition to the core components that make up a Sieve application, application developers can also create other components that interact with the `Workbench` to further define the application's behavior. In this section we look at components that provide editors for `WorkbenchComponent` objects and mechanisms for selecting tools.

5.6.1 The `WorkbenchComponentEditor` Interface

The `WorkbenchComponentEditor` interface (Figure 5.15) provides a minimal definition of an editor for arbitrary objects. The interface is simple: Editors need only be able to accept a `WorkbenchComponent` instance as a “target” for editing.

The default implementation of this interface provides a BeanBox-style property sheet. Application designers may, however, provide alternate implementations of the interface to handle application-specific components in special ways.

The default implementation works by first using the `WorkbenchComponent` instance's `getModel()` method to obtain a reference to the component's underlying object. The JavaBeans introspection

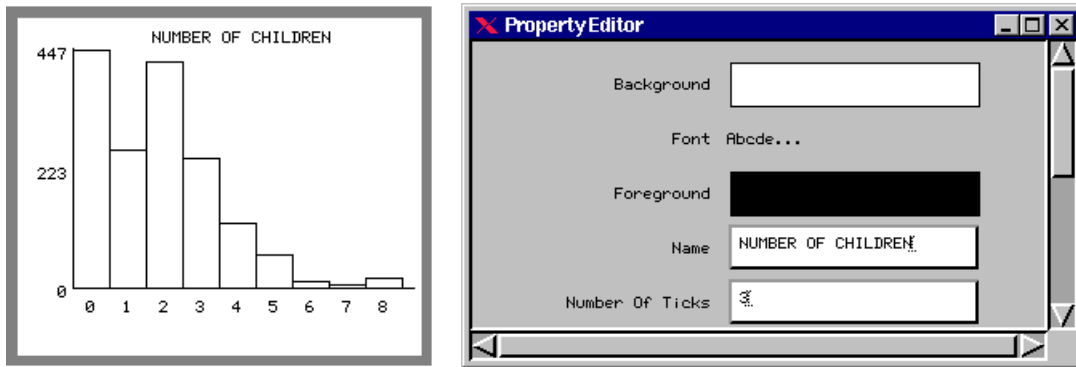


Figure 5.16: A component and its property editor.

mechanism is then used to find all pairs of `get` and `set` methods that conform to the JavaBeans property pattern. For each property, an attempt is made to locate an editor for the property type. (The JavaBeans specification explains the mechanism used to find specific property editors.) All property editors are then presented in a scrolling list.

Figure 5.16 shows a simple histogram component and the automatically-generated property editor used to manipulate it. In the property editor, the user may click on the blocks of color next to “Background” and “Foreground” to change the background color or the color in which the bars are drawn. The name displayed on the histogram may be changed by editing the “Name” field in the property editor, and the number of tick marks shown on the vertical axis may be changed by editing the “Number Of Ticks” field.

In our present implementation, each `Workbench` object supports registration of a single `WorkbenchComponentEditor` instance as the “current editor” for objects on the workspace. When a `WorkbenchComponent` requests an editor, the registered `WorkbenchComponentEditor` instance will be used. Application-specific components may, of course, provide alternate means of presenting an editor to the user.

One motivation for implementing a different `WorkbenchComponentEditor` would be to provide access control. For example, an application’s editor might check a configuration setting to determine if the current user is an “expert.” Novice users could then be provided with access to only a subset of available properties, while expert users could access all attributes of a component. Other applications might provide an editor with the ability to edit components based on some application-specific pattern other than the JavaBeans property pattern.

`WorkbenchComponentEditor` implementations also have the opportunity to detect changes made by the user to properties that were not implemented as bound properties (i.e., cases where the object does not fire a `PropertyChangeEvent` when a property is modified). This situation may arise for a number of reasons, perhaps most commonly when the component inherits from classes such as `java.awt.Component` whose properties are not normally bound. Under the component editing model described in Section 5.2.3, the `Workbench` would not normally be notified when these

properties are changed. As we describe later, this would be problematic for components that are being used collaboratively.

To help prevent these kinds of “unnoticed” changes, `WorkbenchComponentEditor` implementations may also invoke the `Workbench` instance’s `componentEdited` method whenever, for example, a property is changed via a property panel. It is the responsibility of the `Workbench` to ignore duplicate notifications if the property change is also reported by a `WorkbenchComponent`. For example, a button class might expose its label as a bound property. When added to the Sieve workspace, an instance of this class would be wrapped in Sieve’s default `WorkbenchComponent` implementation, which would register itself with the button as a `PropertyChangeListener`. If the user selects the button, the `WorkbenchComponent` will request that Sieve present the user with a property editor (an implementation of the `WorkbenchComponentEditor` interface). When the user edits the button’s label using the property editor, both the `WorkbenchComponentEditor` and the button’s `WorkbenchComponent` can detect the change and report it to the `Workbench`. The `Workbench` will determine that the second notification can be ignored.

Allowing `WorkbenchComponentEditor` implementations to report changes to the `Workbench` increases the likelihood that an arbitrary bean can be used collaboratively in Sieve. Instead of requiring that the state of a collaboration-unaware component be exposed as bound properties, any state information exposed as *unbound* properties can also be shared, provided these properties are edited through the `WorkbenchComponentEditor`.

5.6.2 Toolboxes

Most Sieve-based applications will have some kind of “toolbox” – an object in the user interface that permits selection of a tool for adding a component to the workspace. Applications can have more than one such object.

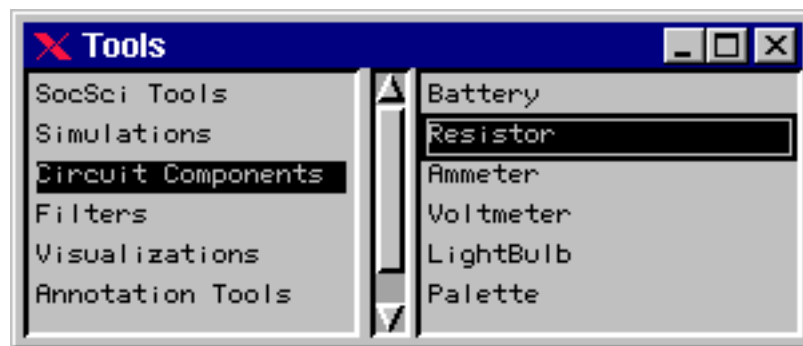


Figure 5.17: A hierarchical toolbox.

Sieve provides support for building toolboxes, but does not place restrictions on how these objects are implemented. There is, for example, no requirement that they conform to a given interface. Essentially, any object with a reference to a `Workbench` instance can tell that `Workbench` what its current tool should be by invoking the `Workbench` instance’s `setCurrentTool` method.

In general, toolboxes have three functions: show the user what the available tools are, let the user choose a tool, and show the user what the current tool is.

Figures 5.17 and 5.18 show two possible means for displaying the available tools. The hierarchical toolbox in Figure 5.17 displays the names of sets of tools in the left column, and individual tools from the selected set in the right column. The toolbar-style component in Figure 5.18 displays buttons for a fixed set of whiteboard tools. (The “Tools” menu in Figure 5.18 is yet another toolbox implementation.)

In these two examples, tool selection is accomplished by clicking on the name or icon for the desired tool. The currently-selected tool is indicated by a highlighted name or icon. In both cases, clicking on the name or icon of the selected tool before creating a component de-selects the tool.



Figure 5.18: A toolbar for selecting whiteboard tools.

These two figures also illustrate two types of toolboxes. The hierarchical toolbox is a general-purpose implementation, capable of providing access to arbitrary tools. The toolbar, however, is more application-specific. It provides access to a fixed, specific set of tools.

Application-specific toolboxes will typically instantiate `Tool` objects directly or access specific pre-existing `Tool` instances. General-purpose toolbox implementations are more likely to use mechanisms like the `ToolFactory` (described in Section 5.3) to map a component class to a specific `Tool` implementation.

ToolListener	
Purpose	Registered <code>ToolListener</code> objects are notified when a <code>Workbench</code> instance’s current tool is changed, when a tool is used to create a component, and when the “tool retained” property is changed.
Methods	<pre>public abstract void toolChanged(ToolEvent e) public abstract void toolUsed(ToolEvent e) public abstract void toolRetainedStateChanged(ToolEvent e)</pre>

Figure 5.19: The `ToolListener` interface.

For applications with more than one toolbox, the `ToolListener` interface (shown in Figure 5.19) provides support for coordinating between toolboxes. For example, if the hierarchical toolbox and toolbar shown in Figures 5.17 and 5.18 are part of the same application, each can implement the `ToolListener` interface and listen for tool changes on a `Workbench` instance. This primarily serves as a means for allowing each toolbox to keep its display consistent with the state of the currently-selected tool. If the user first chooses a tool in one toolbox and then chooses another tool in the other toolbox, the first tool can be un-highlighted.

5.7 Summary

In Chapter 4 we described eight objectives for our design of a collaborative system. The components presented in this chapter address six of these objectives.

1. **Flexible component construction** is achieved by allowing developers to write new implementations of the `Tool` interface and register them with the `ToolFactory`.
2. **Flexible component linking** is achieved by allowing new implementations of the `Link` interface to be written. These may be registered with the `LinkFactory`.
3. **Flexible component selection** is achieved by allowing developers to write new `WorkbenchComponent` implementations that have complete control over how the user interacts to achieve selection, how selection is indicated, and whether or not an editor is presented for the selected component.
4. **Flexible component display** is also supported by allowing creation of new `WorkbenchComponent` implementations which display components in different ways.
5. **Support for aggregate components** is provided by allowing custom `WorkbenchComponent` implementations to provide editors for subcomponents. Custom `WorkbenchComponent` implementations can also determine how links to subcomponents are to be initiated and terminated. Custom `Link` implementations can also provide encapsulated state information describing how subcomponents are linked.
6. **Minimal application development overhead** is achieved for applications that need `BeanBox`-like behavior by providing default implementations of the `Tool` and `WorkbenchComponent` interfaces. These can be used to add arbitrary objects to the Sieve workspace and manipulate them in much the same way that they would be manipulated in the `BeanBox`.

The two remaining objectives (decoupling of collaboration support and support for writing collaboration-aware components) will be covered in Chapter 6.

Chapter 6

Collaboration Support

This chapter describes Sieve’s support for real-time collaboration. Collaboration within a Sieve session may be thought of as occurring at two levels. First, all collaborators see the same workspace structure. They see the same components on the workspace at the same locations, with the same links between them. In addition, each component on the workspace has the same internal state across all sessions.

6.1 Distributed Architecture

Distributed software architectures to support collaboration can be classified within a range of implementations. At one end are centralized architectures, where the shared application is maintained in one physical location and the application display is distributed; at the other are replicated architectures, where the shared application is copied to each collaborator and change notification events (e.g., user input events) are distributed [5]. Replicated architectures generally require less network bandwidth, because only events are distributed instead of graphical display information.

We have chosen to implement Sieve’s collaboration support using a replicated architecture, partly because Internet users may be separated by relatively long network lags on low bandwidth connections. Use of a replicated architecture also allows us to provide location-relaxed WYSIWIS. As discussed in Section 3.1, this allows collaborators to view and manipulate the same or different parts of the workspace. Location-relaxed WYSIWIS is not possible with centralized architectures, since only a single instance of the application exists. The display information generated by this single instance is seen by all collaborators.

Although each Sieve replica contains all workspace functionality, a central server is still used when running Sieve in collaborative mode. As we discuss below, this server is used to simulate multicasting, giving the appearance of being able to send messages to all sessions with a single “send” operation, and it also provides a central location for storing persistent representations of workspace content and state.

6.2 The Java Shared Data API

Messaging between collaborating Sieve workspaces is provided by the Java Shared Data API (JSDA) from Sun [30], an experimental class library designed to support collaborative applications. JSDA allows collaboration-aware Java objects to send data to some or all of the participants within a communications session. JSDA provides an abstraction of a communications session, to which client objects in different Java virtual machines may be attached.

A JSDA session can contain one or more channels. Any client object may send messages over a channel. Messages may be sent to all clients, to a specific client, or to all clients other than the sender. JSDA channels may be specified as reliable, in which case delivery is guaranteed, or unreliable, in which case delivery is not guaranteed. JSDA channels can also optionally enforce uniform message ordering.

6.3 Broadcasters, Controllers, and Servers

To support our objective of decoupling collaboration functionality from basic workspace functionality, collaboration support is provided entirely by objects used in composition with the core Sieve components described in Chapter 5. This allows applications that do not require collaboration to be built without the overhead of excessive unused code. It also allows new collaboration support mechanisms to be written and used with the core Sieve components.

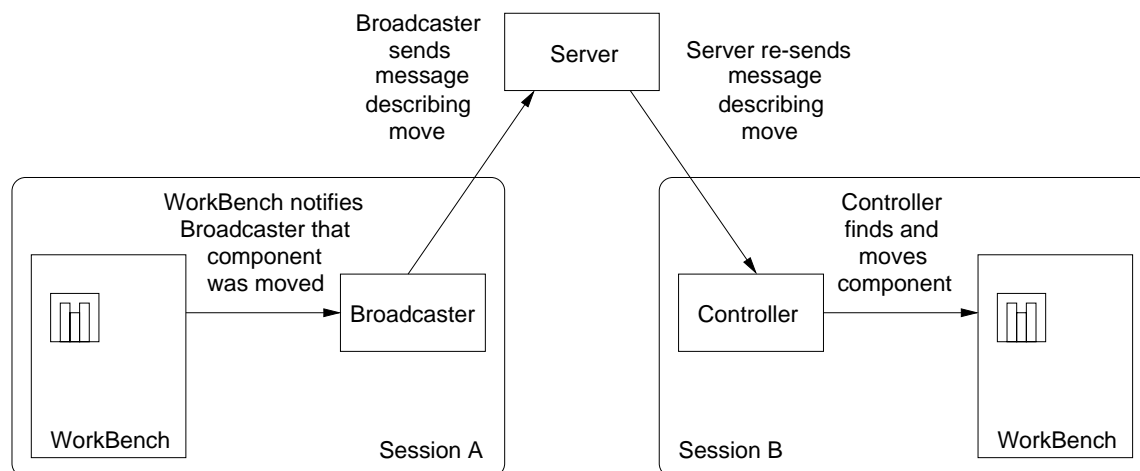


Figure 6.1: High-level interactions between collaborating sessions.

In our current implementation, there are three objects responsible for synchronizing multiple Sieve sessions. In the session that initiates a change to the workspace, a broadcaster object detects the change and constructs a message describing it. This message is then sent to a central server, which distributes the message to all other sessions. The message is received by a controller object in each of the other sessions. Each controller then manipulates its local workspace to reflect the

change described by the message. Figure 6.1 illustrates the interactions between these three types of objects when (as an example) a component in one session’s workspace is moved.

The broadcaster object is implemented by Sieve’s `Broadcaster` class. This class implements the `ToolListener` and `WorkbenchListener` interfaces, and can therefore be attached to a `Workbench` object as a listener for both tool events and workbench events.

The server object is implemented by Sieve’s `Server` class. This class instantiates a JSDA server object whose primary purpose is to relay messages from one session to all other sessions, simulating multicast functionality. The `Server` also implements support for recording Sieve sessions as described in Section 6.6.

<code>WorkbenchController</code>	
Purpose	The <code>WorkbenchController</code> interface defines the behavior of objects that can receive Sieve messages, often for the purpose of updating a <code>Workbench</code> instance based on changes in a collaborating session.
Methods	<pre> public void doAddComponent(Class c1, MouseEvent evts[], UniqueID id) public void doRemoveComponent(UniqueID id) public void doSetBounds(UniqueID id, Rectangle b) public void doChangeProperty(UniqueID objID, PropertyChangeEvent e) public void doMakeLink(Class linkClass, UniqueID srcID, UniqueID destID, UniqueID linkID, Object state) public void doRemoveLink(UniqueID linkID) </pre>

Figure 6.2: The `WorkbenchController` interface.

Controller objects conform to the `WorkbenchController` interface, shown in Figure 6.2. An implementation of this interface is provided by the `Controller` class. The `WorkbenchController` interface includes a method for performing each basic workspace manipulation: adding, editing, moving, and removing components; and adding and removing links between components. The `WorkbenchController` interface essentially mirrors the `WorkbenchListener` interface, described in Section 5.5. `WorkbenchListener` objects are notified when the workspace is manipulated, and `WorkbenchController` objects are able to perform workspace manipulations. (Section 6.6 describes another use for the `WorkbenchController` interface.)

The messages used to propagate changes are subclasses of Sieve’s abstract `Message` class, shown in Figure 6.3. One subclass exists for each type of operation, with each subclass’s `deliver` method invoking the appropriate method of the `WorkbenchController` interface to perform the necessary manipulation of the local workspace.

The `Message` class is an example of the Command design pattern [11]. It encapsulates an operation on the `Workbench` in such a way that the receiver of a `Message` object does not need to know anything about the operation. The receiver simply needs to have a `WorkbenchController` object

Message	
Purpose	The abstract <code>Message</code> class specifies the interface for messages.
Methods	<code>public abstract void deliver(WorkbenchController controller);</code>

Figure 6.3: The `Message` abstract class.

to have the message deliver itself to. Note that the message is not *delivered to* the `WorkbenchController`, but rather the message delivers itself. The `WorkbenchController` is never given a reference to the message. This approach allows new message types to be added easily, since it eliminates the need for the message receiver (the `WorkbenchController`) to perform explicit case analysis on the type of request.

In Sections 5.2 and 5.4, we noted that both `WorkbenchComponent` objects and `Link` objects have `UniqueIDs` associated with them. These are used to identify objects across multiple collaborating sessions. When used in a collaborative application, each `WorkbenchComponent` or `Link` will be assigned a `UniqueID` upon creation.

6.3.1 Handling Component Creation

The `Broadcaster` object begins preparing for component creation when a `toolSelected` event is received from the `Workbench`. At this point, it will register itself as both a `MouseListener` and a `MouseMotionListener`, and begin recording all mouse events generated by the user's actions on the `Workbench`. As described in Section 5.3, this is also what the currently-selected `Tool` object is doing.

When the `Broadcaster` receives a `toolUsed` event, it stops recording mouse events and constructs a message containing the unique identifier of the new component, the class of the new component, and the sequence of recorded mouse events. This message is then broadcast to the other sessions.

When the message is received, the unique id, class, and sequence of mouse events are given to the `WorkbenchController`. The `WorkbenchController` will then find the appropriate tool for the class, replay the sequence of mouse events to re-construct the object, and assign the specified unique identifier to the new object.

6.3.2 Handling Link Creation

For simple connections between components, propagation of link creation is straightforward. When the `Broadcaster` receives a `linkAdded` event, it constructs a message containing the name of the `Link` implementation's class, the unique identifiers of the source and destination components, as well as the `Link`'s unique identifier.

The ability to connect sub-components complicates this somewhat. In Section 5.4, we noted that `Link` implementations may optionally return a block of arbitrary state data from their `getState` method. This block of data can then be passed to a newly-created `Link`'s `init` method. To support connections between sub-components (as well as other types of complex connections), the state data block is also included in the message.

When the message is delivered to a `WorkbenchController` in a collaborating session, it need only instantiate a new `Link` based on the class name specified in the message. It then invokes the new `Link`'s `init` method, passing it references to its source and destination, as well as any state data present in the message. Finally, it sets the unique identifier of the new `Link`.

6.3.3 Handling Other Operations

Message handling for all other workspace manipulations is less complicated. When the `Broadcaster` detects that a `Link` or `WorkbenchComponent` has been removed, it simply creates a message that, when received, instructs the `WorkbenchController` to invoke the `remove` method on the deleted object.

Similarly, when the `Broadcaster` detects that the bounds of a `WorkbenchComponent` has changed, the message it creates will instruct the `WorkbenchController` to move or resize the `WorkbenchComponent` object's visual representation.

Finally, when the `Broadcaster` is notified that a component has been edited, it constructs a message that describes how the `WorkbenchController` should reproduce the change on its local copy of the `WorkbenchComponent`. The ways in which component changes may be described is explained in more detail in Section 6.4.

6.4 Component State Replication

The primary innovation of Sieve's approach to collaboration is the way in which the state of individual components is replicated across multiple collaborating sessions. Specifically, Sieve uses an extended form of property binding to detect when arbitrary components have been edited and then propagate these changes to component replicas in other sessions.

As described in Section 4.1, the JavaBeans specification includes property binding as one of the primary means of communication between objects. In Sieve, we have taken the idea of binding similarly-typed properties of two distinct objects and extended it to support binding of the same property in replicas of an object in different collaborating sessions. A local property binding

mechanism (like the one implemented by BeanBox) preserves consistency between two local objects. Our mechanism preserves consistency between two remote object replicas.

This approach allows components to be used in a collaboration-unaware manner, since any component that follows the JavaBeans specification for bound properties may be shared on the Sieve workspace. Although state changes propagated in this manner tend to be at a high level, this model works well for many kinds of components. Components whose state and configuration may be completely described by a fixed set of small, atomic, independent properties can generally be reliably replicated by propagating changes made to these properties.

Consider, for example, the Sieve-based visualization environment described in Section 3.2. The source, filter, and visualization components used in this environment lend themselves to configuration via properties. Source modules tend to have properties that specify a location of raw data. Filter modules tend to have properties that specify the parameters of the filter operation and indicate which fields are to be filtered. Visualization modules tend to have numerous properties that define exactly how the data are to be presented.

There are, however, cases where replication by property binding is not desirable. There are at least three common categories of components for which this type of replication is not appropriate:

1. **Components that implement editors.** It would, for example, not be desirable to replicate the entire content of a text editing component or a bitmap drawing component after each edit. In addition to simply being inefficient, this approach would generally prevent two users from being able to use the component simultaneously, since one user's changes would often be overwritten by those of another user. The developer would then have to provide some sort of floor control mechanism to prevent more than one user from manipulating this kind of component at any given time.
2. **Components that depend on external entities, such as the system clock.** For example, replicas of a simulation that use the system clock for timing will likely behave at least slightly differently when running on two different machines. Such a simulation might be started when a user clicks a "start" button and stopped when the user clicks a "stop" button. Since the start and stop messages will arrive at different times on each machine, each user's replica of the simulation will have run for a slightly different duration. Restoring the state of these components will also be problematic.
3. **Components whose behavior changes when used collaboratively.** For example, two replicas of a component may need to operate as "master" and "slave," with only the master component actually modifying shared data of some sort. Components that implement games are another example. These will typically have to behave differently for each user. In a multi-player dungeon game, the dungeon as a whole will have the same state for all users. Each user, however, will usually see a different part of the dungeon.

The first problem could almost be solved by the JavaBeans "indexed" property change mechanism. A bean property may be designated as indexed, allowing it to be treated as an array. Rather than being represented as a (name, value) pair, an indexed property is represented as a (name, integer index, value) triple. In a text editor, for example, the index could represent a paragraph or line

number. Unfortunately, the current version of the `java.beans` package does not include support for distinguishing indexed property changes from other property changes. There is, for example, no `IndexedPropertyChangeEvent`.

To solve this problem for the three categories for components listed above, we rely on a higher-level abstraction of a component change: the `EditEvent` class described in Section 5.2.3. Messages describing component changes simply include an `EditEvent` instance. When such a message is received, the controller locates the local replica of the changed component and passes it to the `EditEvent`'s `applyChange` method.

In the case of simple property changes to arbitrary components, the `EditEvent` will be an instance of the `PropertyEditEvent` class. As discussed in Section 5.2.3, invoking such an object's `applyChange` method will result in the appropriate property being updated. Components that want to provide more complex forms of editing need only provide their own implementation of an `EditEvent` subclass.

Consider, for example, a shared text-editing component. As noted above, exposing the entire content of the text editor as a single property is likely to be inefficient. Each keystroke will result in a property change, forcing the entire content to be transmitted to all collaborating sessions. As an alternative, the editor could be implemented so that each change resulted in an `EditEvent` being fired. The `EditEvent` could contain only the text that was added or removed, along with appropriate context information. For example, it might simply know that the character “h” was added between the characters at positions 2 and 3 on line 4. The `EditEvent`'s `applyChange` method would know how to update another instance of the text editor by finding the appropriate place in the appropriate line and adding the character.

Using `EditEvent` objects to replicate component state satisfies our objective of allowing components to be written with varying degrees of collaboration awareness. Components that use the JavaBeans bound property mechanism can be used in a collaboration-unaware manner. Where this is not appropriate, collaboration-aware components can also be written, with custom `EditEvent` implementations used to encapsulate component-specific protocols.

6.5 Preventing Message “Echo”

To ensure replicated behavior across all collaborating sessions, it is necessary that the `Workbench` fire appropriate events to any registered `WorkbenchListener` objects for actions performed by remote users, just as it would for actions performed by local users. Since the session's broadcaster object is among the registered `WorkbenchListeners`, the opportunity exists to create message “echo” – messages being generated based on the actions of remote users.

The changing of a property is perhaps the most obvious situation in which this can occur. When the controller object receives a message specifying that a property change is to be made, it finds the appropriate component, obtains a reference to the property's “setter” method, and invokes that method with the new value. If the changed property was a bound property, the component will fire a property change event. This event will eventually be passed up to the `Workbench`, which will generate a `componentEdited` event and send it to registered `WorkbenchListener` objects.

Normally, the broadcaster would detect this event and send a message describing the change to all other sessions.

This type of echo will usually die quickly. By convention, JavaBeans are supposed to ignore duplicate property changes. Hence when the duplicate message arrives, the component would find that its property was already set to the specified value and would therefore not fire a property change event. It is, however, possible to produce an echo that will eventually halt the system if two users change a bound property to different values at roughly the same time. If properly timed, an incoming property change message with a new value can arrive immediately after a property change has taken place. This scenario can easily result in a large number of new messages being introduced into the system.

A similar but more subtle problem can arise from resizing a component. Since Sieve supports placement of arbitrary components on the workspace, each component can take responsibility for its own layout and therefore for its own size. Our default `WorkbenchComponent` implementation, for example, simply asks a component's visual representation for its preferred size. The wrapper around the component is then sized accordingly. When resizing such a component, the wrapped object may choose to restrict its maximum size. The result is that a given resize action can result in multiple resizings, e.g., first to the size indicated by the user and then to the size preferred by the component.

Normally, the only side effect of this scenario is that every session attempts to process each resize event. Extra messages may be introduced, but eventually each session's replica of the component will have the correct size. The scenario becomes complicated, however, if the wrapped component is a platform-dependent native component and sessions are running on two or more different platforms. In these cases, the preferred size of such a component may vary widely from platform to platform. This means that there is no "final" size for the component across the entire session.

Consider, for example, a `java.awt.Choice` (pop-up menu) object. Such an object (with the same contents) might take 75 horizontal pixels under Windows 95, and 100 horizontal pixels under UNIX and X-Windows. If the user on the Windows machine attempts to resize the choice box to 120 pixels, the component will snap back to 75 pixels and fire a "bounds change" message indicating a width of 75 pixels. (Depending on the choice box implementation, this may or may not happen immediately. If there is a delay, then another message indicating a width of 120 pixels might be sent prior to this message.)

When the session running under X-Windows receives this message, it will attempt to change the width of the choice box to 75 pixels. Since this is smaller than the object's preferred size, it will snap back to 100 pixels, resulting in a bounds change message being sent with this size. Under these circumstances, the two sessions will "fight" over the size of the component indefinitely.

To solve these problems, we simply have the broadcaster contact the controller to see if a given event from the `Workbench` could have been generated as a result of the message that the controller is currently processing. If, for example, the broadcaster receives an event indicating that a component has been edited, it first checks to see if the property change described by the event matches a property change message currently being processed by the controller. (To generalize the process of checking, `EditEvent` objects are compared. The broadcaster and controller never explicitly

examine property information.) This coupling between broadcaster and controller is undesirable, but is necessary if arbitrary components are to be supported on the Sieve workspace.

This approach is still somewhat fallible. It assumes, for instance, that the event fired by the component does in fact match the event described in the incoming message. A malicious component could, for example, fire an event claiming that a property had been set to a random value. The resulting echo would likely bring down all sessions. Since Sieve must treat all components as opaque, there is also no way to determine with complete certainty why a property change event was fired. It is also not possible for Sieve to simply prevent components from firing property change events for changes made remotely, since these events are generated within the components themselves. Even if it were possible to block the generation of these events, it would likely not be desirable, since other components on the workspace could be listening for property change events.

6.6 Storing Workspace State

Storing the state of the shared workspace serves two functions. It allows users to work asynchronously, since collaborators need not be present in the workspace at the same time. One user may begin working, leave notes or other annotations for another user, and then exit the workspace. After all users exit the workspace, the next user to enter the workspace should see the same workspace state as the last user that left. Persistence also allows “late-joiners” to enter the workspace once one or more collaborators are already present. In this case, the most recent user to enter the workspace should see the same workspace state as the other users. In both cases, it is necessary to transfer the state of the shared workspace from the server to a new client upon connection.

Lauwers and Lantz [22] describe three approaches for implementing state transfer in collaborative software:

1. A history of the events that led to the current state can be replayed to the late-joiner.
2. State information can be uploaded from an existing client and downloaded to the late-joiner.
3. Process migration techniques can be used to copy an existing client to a late-joiner. These techniques typically suspend an executing process and take a snapshot of the state and behavior [2]. The snapshot can then be restarted at a later time or on a different machine.

Each of these approaches has distinct advantages and disadvantages. Event replay is relatively straightforward, but the event history may become impractically large for long sessions. It may also be problematic if the state of the workspace depends on external factors, such as system clock values or random number generators. Uploading current state information can solve these problems, but requires that each shared entity know how to encapsulate its state in a transferable way and then initialize itself from this encapsulated information. Process migration techniques attempt to automate this state transfer, but problems arise if the process accesses any local resources (such as a file) that may not be accessible when the process is restarted.

We have chosen to use an event replay mechanism to provide persistence for Sieve workspaces. The use of a state-transfer mechanism was rejected simply because it would prevent arbitrary components from being used, since all components would have to implement methods to produce a state representation and then initialize themselves from that state information. Process migration (implemented with Java's object serialization capabilities) remains a possibility for future versions. For serialization of the entire workspace state to be successful, however, all components would have to be serializable. Many simple components could be serialized automatically by Java's default serialization mechanism, but any moderately complex component would have to explicitly override the `readObject` and `writeObject` methods.

All messages are recorded by a special client object attached to each channel by the Sieve server. The client implements the `ChannelRecorder` interface, which extends the `WorkbenchController` interface. This allows messages to be delivered to a `ChannelRecorder` in the same way that they would be delivered to any other `WorkbenchController` implementation. In addition to the behavior specified by `WorkbenchController`, `ChannelRecorder` implementations also provide means for retrieving an array of stored messages. When a new Sieve client joins the session, it can simply contact the `ChannelRecorder`, retrieve the array of messages, and replay them.

We have built two implementations of the `ChannelRecorder` interface. Our earliest implementation provides full replay of (almost) all messages that occur on a channel. To make this slightly more efficient, messages representing telepointer movement are not stored. In addition, only the last bounds change message for each component is kept.

Our second implementation attempts to provide a more compact sequence of events for reconstructing the state of a workspace. It filters messages as follows:

- Only the most recent bounds change message for each component is stored.
- Only the most recent property change message for each property of each component is stored.
- Any messages that refer to a deleted component are removed.
- Link creation messages are removed when links are deleted.
- The relative ordering of all messages is preserved.

Not surprisingly, the latter implementation provides faster replay. It may, however, prove problematic for complex components. Specifically, components that constrain property values based on other property values might not be restored properly by our minimal-replay approach.

6.7 Conflict Resolution

Since Sieve does not implement a floor control policy, the possibility exists that two users can simultaneously edit a component in incompatible ways. Figure 6.4 illustrates a common example of this type of conflict. In this example, user A changes a component's background from blue to red. Simultaneously, user B changes the same component's background from blue to yellow. The

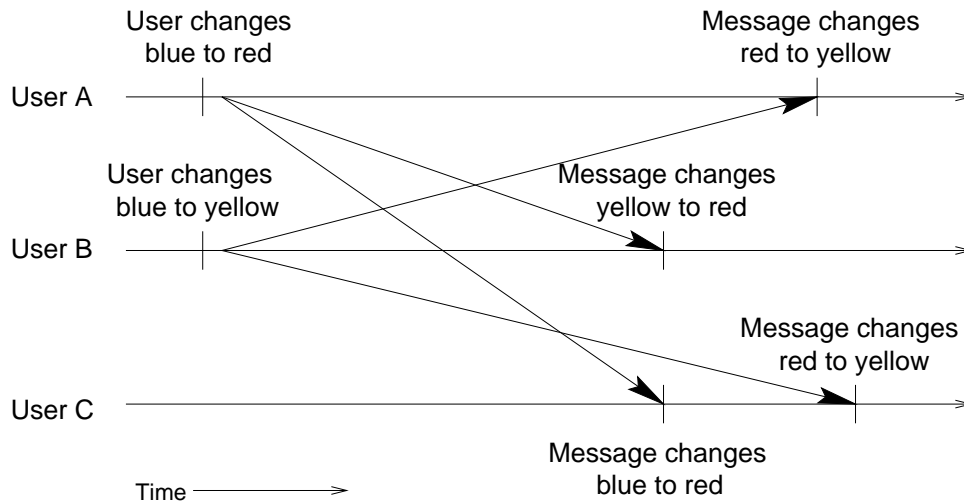


Figure 6.4: Conflict caused by simultaneous edits.

messages describing these changes can arrive in the order shown, resulting in the two users having inconsistent views of the component's state.

JSDA, the class library that Sieve uses to simulate multicast messaging between sessions, will ensure that all sessions receive the messages in the same order. Hence in our example, all *other* sessions (User C's session, for example) will have views of the component's state that are consistent with each other and with either Component A or B.

One solution to this problem would be to send the message describing the change to all sessions, including the one that initiated the change. The actual change of component state could then be delayed until the message describing the change arrives. Since all sessions receive the messages in the same order, consistency would be guaranteed.

This solution, however, would require explicit support by components on the workspace. For example, when a local change was made to a component property, the component would need to fire an event describing the change, but then wait for permission to actually make the change. Since Sieve needs to be able to treat arbitrary objects as "black boxes," it cannot assume or require that components will behave this way.

Another possible solution would make use of global clocks (e.g., Lamport's logical clocks [20] or vector clocks [10]) to detect potential conflicts. If a session received a message that was generated concurrently with a message that it had recently sent, the server's persistent record of the session could be consulted to determine if the new message should be discarded or not.

This approach has a number of drawbacks. Maintaining global clock information increases the size and complexity of all messages. In order to detect concurrent messages, each session must store some number of recent messages for comparison. Finally, the persistence mechanism must be extended to provide support for determining the relative order of two messages. This not only

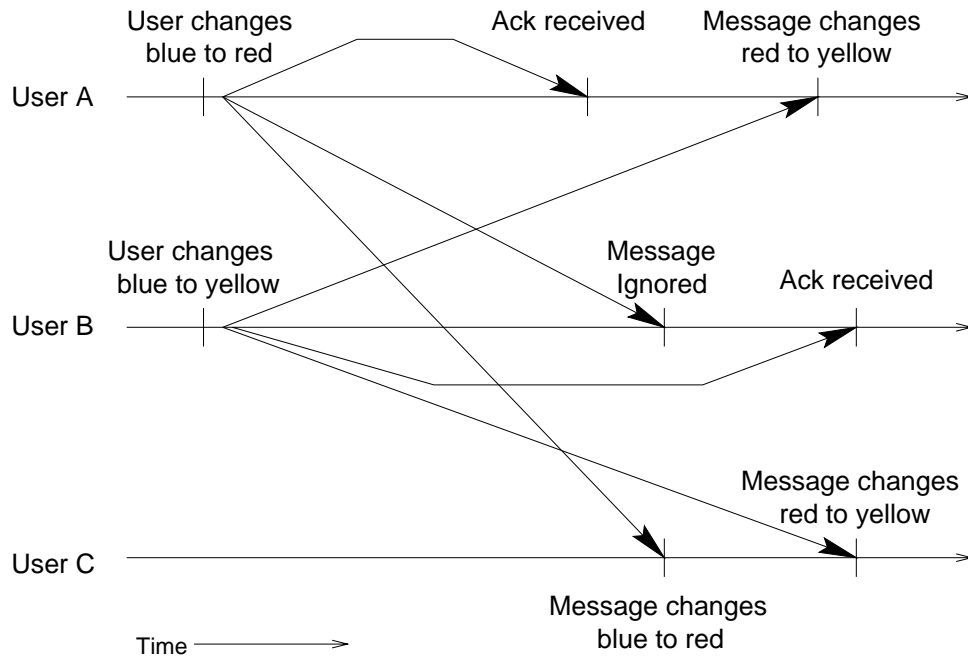


Figure 6.5: Resolving conflicting edit messages.

increases the mechanism's complexity, but also likely eliminates the possibility of implementing persistence in some way other than event replay.

Our solution is something of a hybrid of these two approaches. Upon construction, edit messages (messages that describe changes to a component) are temporarily stored by each session in a vector of “unacknowledged” messages. These messages are then sent to all sessions, including the session that originated the message. When such a message is received by its originator, it is treated as an acknowledgement, removed from the vector of unacknowledged messages, and thrown away. When an edit message originated in a different session is received, it is checked against all unacknowledged edit messages. If any unacknowledged message would obsolete or overwrite the incoming message, the new message is thrown away.

Figure 6.5 shows how this approach solves the problem illustrated in Figure 6.4. User A receives acknowledgement of the change he originated prior to receiving the change from User B, so User B's message is delivered. User B, on the other hand, does not receive acknowledgement of his change prior to receiving the message from User A, so User A's message is ignored. After all messages have arrived and been processed, all three clients have consistent views of the component's state.

6.8 The Participant WorkbenchComponent Implementation

As described in Section 3.1, knowledge of remote participants' actions and locations are provided through two mechanisms: telepointers and a multi-user radar view. The former displays a repre-

WorkbenchParticipant
<p>Purpose</p> <p>The <code>WorkbenchParticipant</code> class provides a representation of a Sieve participant's name, color, mouse pointer location, and viewport bounds. It implements the <code>WorkbenchComponent</code> interface and is therefore directly shareable across all collaborating sessions.</p>
<p>Methods</p> <pre> public String getName() public void setName(String name) public Color getColor() public void setColor(Color color) public Point getLocation() public void setLocation(Point location) public Rectangle getBounds() public void setBounds(Rectangle bounds) </pre>

Figure 6.6: The `WorkbenchParticipant` class.

sensation of each user's mouse pointer, while the latter indicates the part of the workspace that each user is currently viewing.

In our initial BeanBox-based prototype, support for sharing participant information was implemented separately from support for propagating workspace changes. Our current design, however, takes advantage of Sieve's `WorkbenchComponent` and `Tool` interfaces to eliminate the need for implementing specific support for awareness mechanisms.

The `WorkbenchParticipant` class, shown in Figure 6.6, implements the `WorkbenchComponent` interface and exposes bound properties for the user's name, color, mouse location, and viewport bounds. A change in any of these properties is propagated to all collaborating sessions just like a property change in any other type of `WorkbenchComponent`. A `WorkbenchParticipant` object is itself invisible, but provides a telepointer component as its visual representation.

An instance of `WorkbenchParticipant`, representing the local user, is created when a client successfully logs into a Sieve session. Typically, the visual representation of this component is not added to the workspace. Instances of `WorkbenchParticipant` representing remote users are created when the `WorkbenchController` is instructed to do so, i.e., when other users log in. The `ParticipantTool` class is registered as the tool for creating `WorkbenchParticipant` instances, and it therefore used by the `WorkbenchController` to create `WorkbenchParticipant` objects for remote users.

6.9 The CollaborationSupport class

To simplify the process of constructing a Sieve client applet or application, we have implemented a simple `CollaborationSupport` class that hides the details of attaching a Sieve `Workbench` object to a collaborative session.

CollaborationSupport	
Purpose	The <code>CollaborationSupport</code> class hides the details of setting up a Sieve client.
Methods	<pre> public static String[] getSessions(String host) public static String[] getChannels(String host, String session) public CollaborationSupport(Workbench wb, String host, String session, String channel, String user, Color userColor) public void restoreWorkbench() public MURadarPane getRadarPane() public void shutdown() </pre>

Figure 6.7: The `CollaborationSupport` class.

Figure 6.7 shows the interface of the `CollaborationSupport` class. A client applet or application can use the static `getSessions` and `getChannels` methods to determine what sessions and channels are available on a specified host. Available sessions and channels may then be presented to the user for selection if necessary. Figure 6.8, for example, shows the use of a combination choice box and text field. The user may either type the name of a new channel or choose an existing channel. Clients may also choose to determine session and channel names in any other way that they choose. For example, this information may be taken from configuration files or applet parameters.

The client can then connect a `Workbench` instance to a given session and channel by instantiating a `CollaborationSupport` object and passing it this information, along with a user name and a preferred color for the user. If a session or channel with the specified name does not exist, it will be created by the server.

Once the `Workbench` has been made visible, the applet or application can call `restoreWorkbench` to restore any components and links that had previously been added to the workspace. Any messages stored by the server (using one of the persistence mechanisms described in Section 6.6) will be retrieved and replayed to the `Workbench`. If a multi-user radar pane (such as the one described in Section 3.2) is needed, invoking the `getRadarPane` method will create one if necessary and return a reference to it. Finally, the `shutdown` method will handle the details of cleanly logging out of a Sieve session.

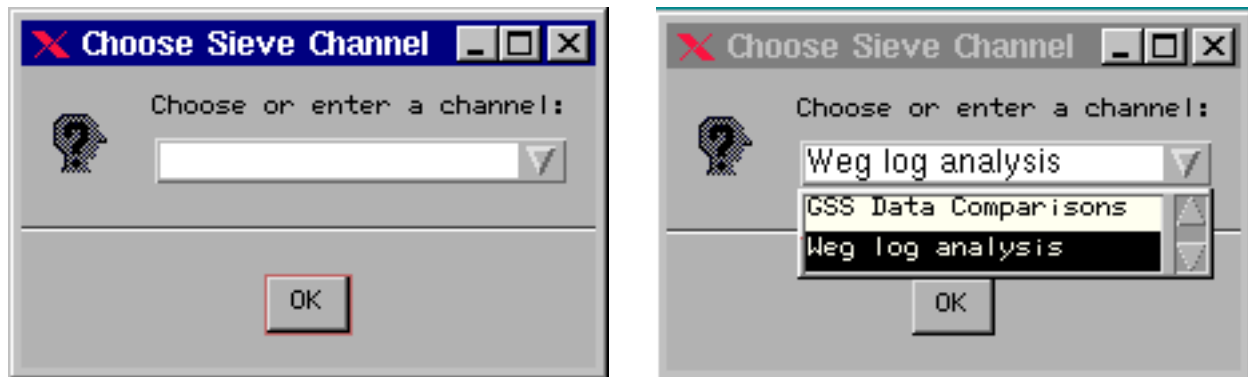


Figure 6.8: Dialog boxes for choosing a channel.

The `CollaborationSupport` class is an example of the Facade design pattern [11]. It provides a simple interface to a complex subsystem, providing the functionality that most client applets or applications are likely to need. More sophisticated applets or applications may, however, still use the subsystem classes (`Broadcaster`, `Controller`, etc...) directly if necessary.

6.10 Summary

Of the eight objectives discussed in Chapter 4, six were addressed by the components described in Chapter 5. The remaining two were addressed by components presented in this chapter.

1. **Decoupling of collaboration support** is achieved by implementing all collaboration functionality in classes that are attached to (or otherwise composed with) Sieve's "core" components. In particular, the `Broadcaster` class is implemented as a `WorkbenchListener`, so the `Workbench` requires no explicit support for generating messages describing changes to the workspace.
2. **Support for writing collaboration-aware components** is provided by the collaboration mechanism's use of `EditEvent` objects to propagate changes. The design of these objects allows developers to write custom `EditEvent` subclasses that implement state replication, or any other type of communication between component replicas, in any way that is needed.

Chapter 7

Sieve Application Design

In Chapter 3 we described several Sieve-based applications at the level of the user interface. In this chapter we examine the design of the classes that provide those applications' functionality.

7.1 Visualization and Dataflow Component Design

This section describes the design of objects used in the Sieve-based collaborative visualization environment discussed in Section 3.2. We begin by describing the dataflow API used by this application. We conclude by describing the implementations of the core Sieve components that allow objects conforming to our dataflow API to be used collaboratively on the Sieve workspace.

7.1.1 Dataflow Network Structure

The concept of generating visualizations by constructing networks of filters was originated by modular visualization environments such as AVS [39]. With these tools, a user can specify a series of operations to be performed on a dataset by connecting icons representing data sources, filtering operations, and visualizations. This network of sources, filters, and visualizations is then executed to produce output.

The intent of our visualization tool differed slightly, in that we wanted to make the dataflow network more interactive by eliminating the distinction between network creation and network execution. This would allow, for example, an appropriate visualization module attached at any point on the network to be instantly updated. Similarly, reconfiguring a filter module would result in the changes being immediately propagated to other network modules.

With the exception of modules which produce or import data, each module in the network has at least one source from which it obtains data. Each module can then act as a data source for an arbitrary number of modules. Figure 7.1 illustrates a simple dataflow network. In this example, module A would either generate data or obtain it from some external source such as a file. Module

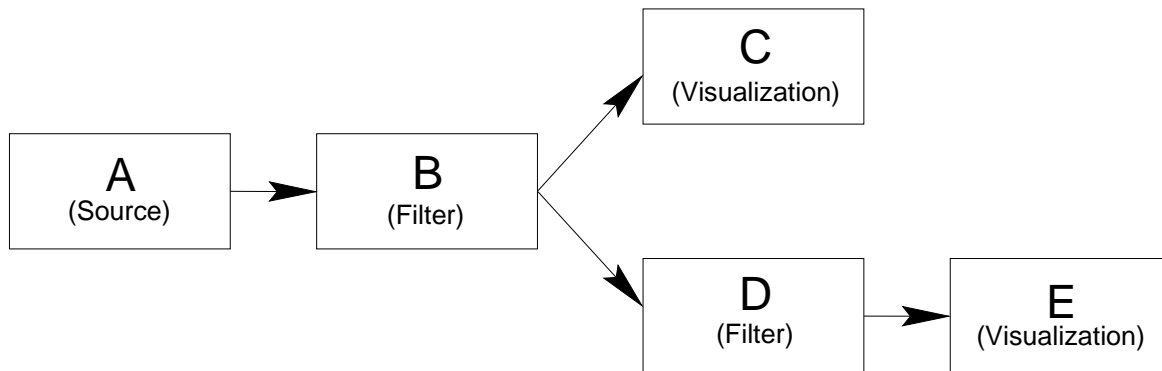


Figure 7.1: A simple dataflow network.

A serves as the source for module B, which serves as the source for modules C and D. Module D then serves as the source for module E.

To make the network interactive, each module must be able to request data from its source at any time. When a module receives a request for data, it returns the requested data if it is stored locally to that module. Otherwise, the request is delegated to the module’s source. In Figure 7.1, for example, if module E requests data from module D, module D could delegate this request to module B. Module B could then pass the request along to module A if necessary.

Allowing requests to be delegated backwards through the network in this manner allows newly-added modules to obtain data immediately. This does not, however, solve the problem of updating a consumer module when its source changes. To handle this scenario, we allow events describing changes to the data to be propagated from any module to its consumers. For example, if module B is a filter that can be configured by the user, any change to B’s configuration could alter the data that modules C and D are using. In this case, module B can fire an event describing the change to modules C and D. These modules can determine if they are affected by the event and re-request data from module B if necessary. If the change affects module D in any way, it can send an event to module E. In this way, a change made to any module in a dataflow network will be propagated to all potentially affected modules.

7.1.2 The TableView Interface

To support development of components for the visualization environment, we developed a simple Java interface to be implemented by nodes in a dataflow network: the `TableView` interface. A `TableView` is a “view” of a table of data. It presents an interface for accessing a two-dimensional table of arbitrary Java objects. It provides methods for discovering both the structure (e.g., dimensions) of the table and also meta-data describing the nature of the objects in the table. Meta-data includes information on the objects’ type and level (interval, ratio, ordinal, or nominal), as well as human-readable information such as field labels and units. `TableView` differs from more generic structures like two-dimensional arrays or dictionaries, which do not typically provide meta-data describing their content.

TableView
<p>Purpose</p> <p>A <code>TableView</code> is a “view” of a two-dimensional table of data.</p>
<p>Methods</p> <pre> public void addSourceView(TableView source) public void removeSourceView(TableView source) public TableDescriptor getTableDescriptor() public String[] getFieldNames() public FieldDescriptor getFieldDescriptor(String field) public int getRecordCount() public Object[][] get(String fields[], int startingAt, int numberOfRecords) public Object[][] get(String fields[], int startingAt) public void addTableViewListener(TableViewListener listener) public void removeTableViewListener(TableViewListener listener) </pre>

Figure 7.2: The `TableView` interface.

The data presented by a `TableView` is logically structured as a set of named fields and a number of records, where each record contains values for the various fields. Note, however, that since `TableView` is just an interface definition, it does not specify or restrict implementation details such as storage format or order.

Figure 7.2 shows the methods that must be implemented by `TableView` objects. The `addSourceView` and `removeSourceView` methods allow `TableView` objects to be linked together. The `getTableDescriptor` method returns an object that provides meta-data describing the table.

Fields are referenced by name. Method `getFieldNames` returns the names of all of the fields that a `TableView` makes available. Each field has a descriptor object, which is returned by the `getFieldDescriptor` method. This descriptor object contains meta-data describing the field, including information such as the type of objects contained in the field and a longer, “human-readable” name for the field. Method `getRecordCount` returns the total number of records in the `TableView`. Data objects are retrieved by calling one of the `get` methods. These methods return a contiguous block of data from the table. Finally, the `addTableViewListener` and `removeTableViewListener` methods are used to register and remove listener objects. These are described in more detail below.

`TableView` objects are in many ways analogous to instances of Java’s `java.io.FilterInputStream` classes. As we describe the usage of `TableView` objects in the following sections, we provide analogies to `FilterInputStream` usage for readers who are familiar with using Java streams. It should be noted, however, that `TableView` objects (unlike `InputStream` objects) provide random access to data.

Types of TableView Implementations

TableView implementations can be divided into three overlapping categories: sources, filters, and consumers.

Sources obtain data from some non-TableView source and present it as a TableView. That source may be something truly external, such as a file or database, or may simply be some other type of Java object. A simple example of the latter case would be a TableView that provides access to a `java.util.Dictionary` object or a two-dimensional array of objects. TableView implementations that act as sources may or may not allow editing. As such, editing functionality is not included in the TableView interface definition.

TableView objects that act as filters take one or more TableView objects as their source and present a view of their source(s) that has been modified in some useful way. These implementations will typically fall into one of several broad sub-categories:

- **Filters that change meta-data.** For example, one type of view could assign colors to fields.
- **Filters that reduce the exposed structure of the table.** These will typically hide fields or records that do not match a given criteria. For example, a view may hide all non-numeric fields in a table. We use the term “hide” rather than “remove” because these filters often do not actually modify data. A TableView can, for example, effectively make a field disappear by simply not including it in the array of field names returned by the `getFieldNames` method. (This is roughly analogous to a `FilterInputStream` object truncating a stream by returning zero from the `available` method. The user of this `FilterInputStream` sees a shorter stream, but the underlying data that the stream was generated from hasn’t changed.)
- **Filters that translate individual objects in the table.** For example, a TableView might replace numeric representations for categorical data with the appropriate human-readable strings.
- **Filters that translate entire tables into new tables.** For example, a filter could translate a table of observations into a table of frequencies. Another filter might interpolate numeric values in a table to produce a larger table, possibly with different *types* of data (e.g., interpolating integer values may require conversion to floating point values).

As with Java stream objects (`FilterInputStream` implementations, in particular), TableView objects are intended to be connected to each other. A TableView object can be connected to another TableView object, allowing complex filtering operations to be constructed by chaining lightweight TableView filters. Figure 7.3 shows how simple filters can be connected to produce data for a histogram. In this example, the raw data consists of records describing people. The first filter passes only those records for people whose age is greater than 20. The second filter processes this information to produce a table of frequency data, which can then be used to produce a histogram.

The final broad category of TableView objects are those that consume data. Typically, these objects would produce some sort of visualization (or other external representation) of the data.

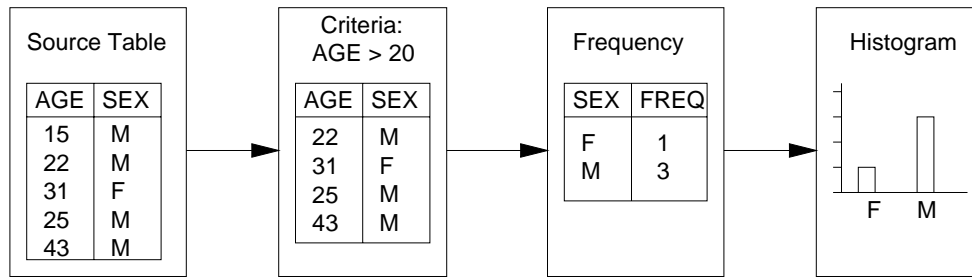


Figure 7.3: A series of manipulations on a table of data.

There is no inherent requirement that these kinds of objects actually implement the `TableView` interface, since they will often simply consume data. This is analogous to streams: Only other `InputStream` objects may act as sources for a `FilterInputStream` but any object may use a `FilterInputStream` to retrieve data.

There are, however, advantages to having visualizations also act as filters. At a minimum, this allows a visualization to appear in the middle of a chain of `TableView` objects, even if it does nothing beyond passing the view of the table provided by its source. In other cases, a visualization might also act as a filter in a more active way, possibly by allowing a user to choose a subset of the visualized data to be processed by filters and consumers appearing later in the chain. For example, a scatterplot visualization may allow the user to select, or “brush,” plotted values. Such a component might then pass only the records containing the selected values to components further down the chain.

Data Push vs. Data Pull

Our design of the `TableView` interface supports pushing events and pulling data. Non-source modules in the network may receive change notifications from their sources. To retrieve data, modules must explicitly request data from their source. An alternative approach would have been a “pure push” model, in which the events used to notify downstream modules of changes included the changed data.

The push model appears somewhat simpler, in that it only requires a single connection between components. On closer inspection, however, this model has a number of characteristics that made it undesirable for our purposes. The push model is data-driven, with activity initiated when data appears or changes. The pull model is demand-driven, with data requested by modules whenever they need it. (The addition of events to a “pure pull” model also allows data-driven behavior.)

As discussed at the start of this section, we would like for the dataflow network to be active in the sense that any change or addition to the network results in an immediate update. This goal presents a problem for the data-driven nature of the push model. Consider, for example, the effect of connecting a module to the end of a dataflow network. In a pull-based system, the new module can simply request data from its source. In a push-based system, the new component has no way of

requesting that data be sent to it. The new module's source must somehow be forced to generate an "artificial" event to pass data to the new module.

As another example, consider a visualization module that allows user interaction to choose a subset of available data for viewing. For example, a line graph visualization may allow the user to choose two fields to be interpreted as independent and dependent variables interactively. When the user wishes to change which pair of fields (out of several) are to be viewed, the module must obtain the data for the selected pair of fields. In a push-based system, such a module must either store all available data or have some means of telling its source to push a new dataset down the network. In a pull-based system, the visualization always has a mechanism for obtaining any data that it might need.

These types of scenarios are less of a concern in systems that enforce a clear distinction between modes of network construction and network execution. Our intent to produce a system that was as interactive as possible made the pull model a clear choice. For applications whose datasets are too large to be manipulated interactively (given memory and performance constraints), this decision might have to be revisited.

Internal Data Representations

`TableView` is an interface and not a concrete class, so no restrictions are placed on how (or even if) a given `TableView` implementation stores the actual data that its view of the table presents. While the particular space and speed requirements of a given application will often dictate these decisions, the following guidelines have been used in developing our current set of `TableView` classes.

- `TableView` objects that only change meta-data typically do not store any actual data from their source. For example, a `TableView` that changes the color associated with a given field would need only intercept requests for that field's meta-data. Requests for data from the table can simply be passed directly to the `TableView`'s source.
- Implementations that reduce the exposed structure of the table should avoid storing data. This is usually simple if the `TableView` only reduces the number of fields, since the `TableView` can control which fields it exposes by having its `getFields` method return a subset of its source's fields.

If the number of records is also reduced (e.g., based on some pattern-matching criteria), some indication must be stored of which records in the table are to be included in the filtered view. Consider, for example, a `TableView` that produces a sample by only passing every fifth record. Such a filter could re-calculate mappings from the reduced view to the full view with each request for data. A request for the third record could be mapped to the filter's source's tenth record (assuming we begin with index zero). Depending on the application, frequency of access, and the selection function, it might make more sense to either store an array of these mappings or perhaps to store the entire (reduced) set of records.

- Implementations that translate individual objects (e.g., those that convert numeric values to strings) should only store translated values if the translation process is sufficiently complicated. Many such `TableView` objects may be able to perform translations on-the-fly.

- `TableView` implementations that translate entire tables typically have to store the entire translated table. For example, consider a filter that converts a table of observations into a table of frequencies. Such a filter would likely only calculate the frequencies once, rather than recalculating them whenever a request for data is received.
- `TableView` objects that act as “pure” sources should attempt to intelligently cache data, especially if the raw form of the data must be retrieved over a network.

Event Handling

All `TableView` objects support registering `TableViewListener` objects, which are notified when the structure or content of the data presented by the `TableView` changes. All `TableView` objects are themselves `TableViewListeners`, and can therefore be notified when their source views change.

<code>TableViewListener</code>	
Purpose	A <code>TableViewListener</code> can register with a <code>TableView</code> to receive notification when the <code>TableView</code> 's data changes.
Methods	<code>public void tableViewChanged(TableView source)</code>

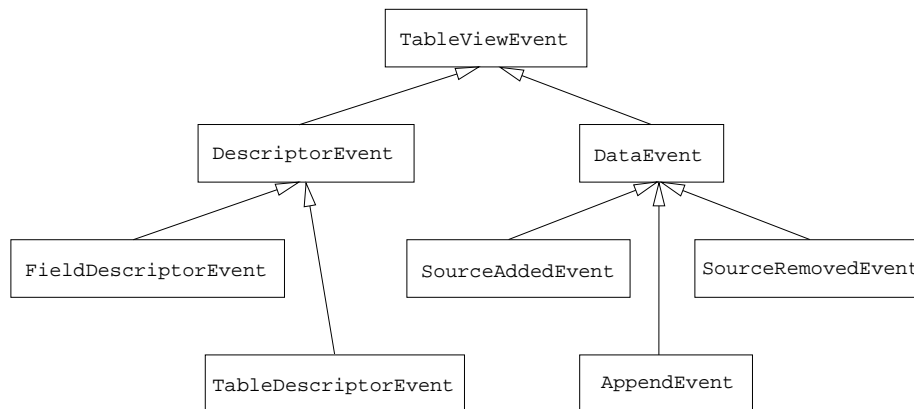
Figure 7.4: The `TableViewListener` interface.

To simplify implementing listener registration, a `TableViewListenerSupport` class is provided that handles adding, removing, and sending events to listeners. `TableView` objects should generate an event any time they change either the content or the structure of the data they present. They must also propagate any events received from their source(s), even if they do not act on these events.

The exception is the case where a given `TableView` object can determine that the event is no longer relevant. For example, consider a `TableView` that has filtered out the field `AGE`. If it then receives notification that `AGE`'s description has changed, that event need not be propagated further.

The current hierarchy of `TableViewEvent` classes for describing different types of changes is shown in Figure 7.5. `DescriptorEvent` instances describe changes to meta-data. `TableView` objects that do not depend on meta-data may simply propagate these events to their listeners. `DataEvent` instances describe changes to the actual data values in the table. `TableView` objects that depend *only* on meta-data can ignore (and propagate) these kinds of messages.

While the `TableViewEvent` subclasses are intended to provide as much detail about the nature of a change as possible, there will always be cases in which a `TableView` receives an event that it does not know how to handle in any specific way. In these cases, the `TableView` should re-initialize itself and send a generic `TableViewEvent` to its listeners. This situation arises when new subclasses

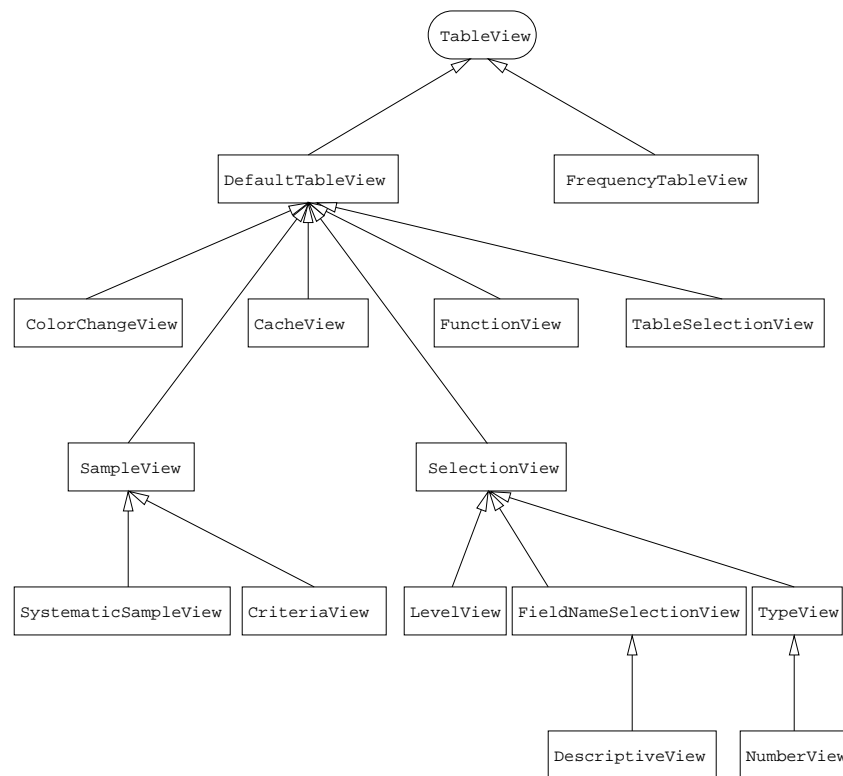
Figure 7.5: `TableViewEvent` hierarchy.

of `TableViewEvent` are introduced, as existing `TableView` implementations will not know how to handle these events in any specific way, and must therefore simply reset themselves.

Current `TableView` Filters

Figure 7.6 shows the hierarchy of our current set of general-purpose `TableView` filters.

- `DefaultTableView` implements a transparent filter. It passes all of its source's data without modification, and also passes all events from its source. It is intended to be used as a base class for more interesting filters.
- `FrequencyTableView` counts the occurrences of each unique value in a single field and produces a table of frequencies.
- `CacheView` implements a simple caching mechanism. On the first data request, it retrieves and stores all available data from its source. The cache is flushed when any `TableViewEvent` is received.
- `ColorChangeView` changes the color assigned to a given field in the table. It does not modify data in the table in any way. (This filter is useful when visualizing data simultaneously in several different ways. Assuming that visualizations use the field color specified by the `TableView`, the data from a given field will be shown in the same color in all visualizations.)
- `FunctionView` applies a function to all values in a field of the table.
- When given a table containing a set of `TableView` objects, `TableSelectionView` allows selection of one `TableView` from the set. All requests for data (or meta-data) are then delegated to the selected `TableView`.
- `SampleView` is an abstract superclass for filters that sample records from a table. Implementations of a `Sampler` interface implement specific sampling strategies.

Figure 7.6: Current `TableView` filter classes.

- `SelectionView` is an abstract superclass for filters that selectively hide and expose fields in a table. Implementations of a `SelectionTest` interface implement field selection strategies. The filtering of fields is accomplished by simply modifying the set of field names returned by the `getFields()` method – no actual modification of the data is required.
- `SystematicSampleView` extracts a systematic sample of records from a table. For example, it might select every fifth record from the table. (This has been implemented primarily as an example – similar classes could be written for other common statistical sampling strategies.)
- `CriteriaView` extracts records that match a particular pattern.
- `FieldNameSelectionView` filters fields based on name.
- `LevelView` filters fields based on the level of the data: interval, ratio, ordinal, or nominal.
- `TypeView` selects fields that contain data of a given type.
- `DescriptiveView` computes descriptive statistics (such as mean, median, mode(s), and variance) for values in a single field of the table.
- `NumberView` selects fields that contain numeric values.

```
public class TableViewTableViewLink extends SimpleLink {
    private TableView sourceView = null;
    private TableView destView = null;

    public void link(WorkbenchComponent source, WorkbenchComponent dest)
        throws LinkException {
        // Cast and save these
        sourceView = (TableView)source.getModel();
        destView = (TableView)dest.getModel();
        // Connect the source and destination
        destView.addSourceView(sourceView);
        sourceView.addTableViewListener(destView);
    }

    public void unlink(WorkbenchComponent source, WorkbenchComponent dest) {
        // Disconnect the source and destination
        destView.removeSourceView(sourceView);
        sourceView.removeTableViewListener(destView);
    }
}
```

Figure 7.7: Implementation of the `TableViewTableViewLink` class.

7.1.3 Supporting TableView Objects in Sieve

All of the `TableView` implementations described above adhere to the JavaBeans specification. All configuration of these objects can be done via bound properties. Customizers for invisible filter and source `TableView` implementations are provided if they contain any non-trivial properties. Otherwise, we rely on automatically-generated property panels.

Support for constructing and presenting `TableView` objects in the Sieve workspace is provided by the default, BeanBox-like implementations of the `Tool` and `WorkbenchComponent` interfaces. The default `Tool` implementation simply places the `TableView`'s visual representation at the point where the user clicks on the workspace. The default `WorkbenchComponent` implementation checks to see if the `TableView` object extends `java.awt.Component`. If it does, then the `TableView` itself is used as the visual representation. If it does not, then either a property editor or customizer is placed on the workspace. As discussed in Section 5.4, the default `WorkbenchComponent` implementation uses the `LinkProxy` and `LinkFactory` classes to determine how a given `TableView` object may be linked.

In the common case of linking two `TableView` objects, the `LinkFactory` locates the class named `TableViewTableViewLink`. Figure 7.7 shows the implementation of this class, with error checking omitted.

7.1.4 Adapting TableView Objects to Other Models

The `TableView` interface provides a generic means for accessing two-dimensional tabular data. As such, it can be easily adapted to similar interfaces used by third-party systems.

The Swing `TableModel`

The `TableModel` interface (shown in Figure 7.8) was introduced in recent versions of Sun’s Swing interface toolkit [37]. This interface provides a description of behavior similar to that of the `TableView` interface. `TableModel` is designed to provide data for spreadsheet-like widgets and similar table-based interface elements.

TableModel	
Purpose	The Swing <code>TableModel</code> interface
Methods	<pre> public int getColumnCount() public String getColumnName(int columnIndex) public Object getColumnIdentifier(int columnIndex) public int getColumnIndex(Object columnIdentifier) public Class getColumnClass(int columnIndex) public int getRowCount() public boolean isCellEditable(int rowIndex, int columnIndex) public Object getValueAt(int rowIndex, int columnIndex) public void setValueAt(Object aValue, int rowIndex, int columnIndex) public void addTableModelListener(TableModelListener l) public void removeTableModelListener(TableModelListener l) </pre>

Figure 7.8: The Swing `TableModel` interface.

`TableModel` is simpler than `TableView`, and is limited in several ways. It provides no direct support for meta-data describing a table or a field in a table. It also only provides support for accessing table data one cell at a time, while `TableView` allows retrieval of “chunks” of data of arbitrary size. Finally, it does not include methods for connecting two `TableModel` instances such that one can retrieve and filter data from the other. While these limitations do not affect `TableModel`’s usefulness for its intended purpose, they do make it a less appealing foundation for constructing efficient filters for use in dataflow networks.

The similarity between the two interfaces does, however, provide an opportunity to implement adapters that convert from one interface to the other. We have written a simple adapter class that allows our `TableView` instances to provide data directly to table-based Swing components. A similar adapter could also be written that allows third-party data sources that conform to the `TableModel` interface to be used in `TableView`-based dataflow networks.

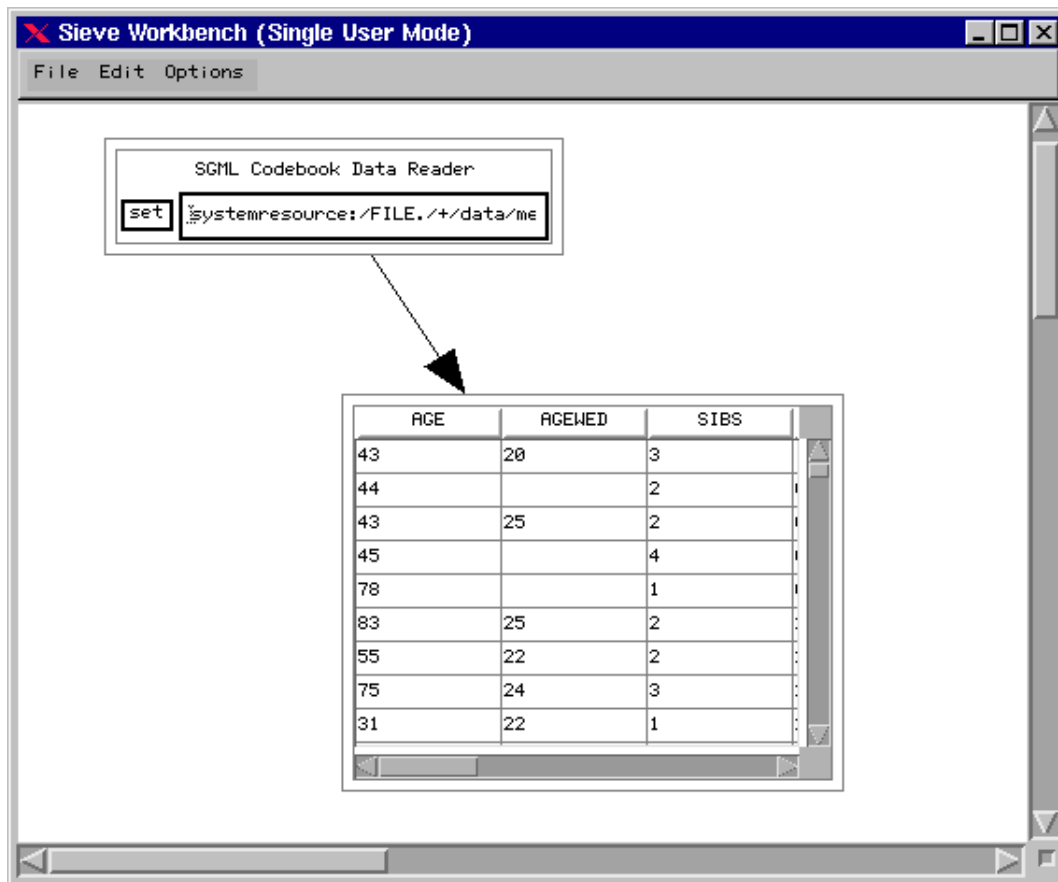


Figure 7.9: A Swing JTable object used in a dataflow network.

Figure 7.9 shows how this adapter can be used to allow a Swing JTable component to be used in a dataflow network. The “SGML Codebook Data Reader” is a TableView implementation that reads data from a file. In this example, we have connected one of these reader objects to a JTable, contained in a simple TableView wrapper object. This wrapper uses our TableModel-to-TableView adapter, allowing it to be connected to other TableView instances on the Sieve workspace.

JavaSpaces and InfoBus

Two emerging technologies – JavaSpaces and InfoBus – promise functionality that is at least similar to that of our TableView-based dataflow architecture. In this section we briefly review the differences and potential interactions between these systems and our dataflow components.

Sun’s forthcoming JavaSpace [36] system will provide a mechanism for distributed data persistence and exchange. It is intended to provide a generic mechanism for storing, querying, and retrieving typed groups of objects, or “entries” in JavaSpace terminology. It supports a somewhat different notion of dataflow from that of our TableView modules, in that it allows distributed algorithms to be

implemented as flows of objects in and out of JavaSpaces. Although still in the early phases of design – only draft specifications are currently available – this system may prove useful as infrastructure for dataflow between remote sources, possibly directed from a Sieve-based application.

InfoBus [33] is a system for dynamic data exchange between JavaBeans. It is being developed by Lotus, and is currently in an early beta release. It provides an abstraction of a bus, to which components may connect in order to exchange information in a structured way. It serves primarily as a broker, negotiating communication between components that produce data with a given structure and components that wish to consume data with a given structure. In this way, InfoBus is conceptually similar to JavaSpace.

While the InfoBus design does allow the creation of data flows between cooperating Beans, the bus design means that all components can exchange information with each other. Implementing a more structured network of communication requires additional logic on top of InfoBus.

In a `TableView`-based dataflow network, communication between adjacent components in the network is direct. Upstream components send events directly to downstream components when something changes, and downstream components directly invoke methods on upstream components to obtain data. The brokering services provided by InfoBus are therefore not needed. As InfoBus technology matures, however, it may be useful to provide the ability to introduce data from external objects into a dataflow network through InfoBus or a similar technology.

7.2 Circuit Simulation Design

The second application implemented for Sieve was the electric circuit simulation described in Section 3.3. This application provides a set of objects representing common components of direct-current circuits: batteries, resistors, light bulbs, ammeters, and voltmeters.

This application is different from our Sieve-based visualization system in several ways. Instead of computational work being performed by individual components, the components serve primarily as user interface elements. The computational work is performed by an invisible `Solver` object, which maintains a matrix describing the current at each point in the circuit.

The `Solver` class (shown in Figure 7.10) represents a circuit as a graph of connected nodes, indexed by Java's `Integer` objects. (A node represents a terminal on a battery, resistor, etc...) Method `addComponent` adds a circuit component with two specific nodes. Method `addNodes` adds a connection (e.g., a wire) between two nodes. Method `removeNodes` removes either a connection between two nodes or the component to which the two nodes are attached. Method `findLoops` solves the matrix and informs each component of the current at each of its nodes.

The solver object is an example of the Singleton design pattern [11]. A single instance of this class exists, created on demand when the first circuit component is added. At that time, a `WorkbenchListener` object is registered with the workbench. This listener object then drives the solver, telling it when and how to update its matrix as components are added, removed, edited, connected, and disconnected.

Solver	
Purpose	The circuit simulation's <code>Solver</code> object maintains a matrix representing circuit connections.
Methods	<pre> public void addNodes(Integer node1, Integer node2) public void addComponent(CircuitComponent newComp, Integer node1, Integer node2) public void removeNodes(Integer SelectedNode, Integer SelectedNode2) public void findLoops() </pre>

Figure 7.10: The `Solver` class.

For example, when the listener receives a `componentAdded` event from the `Workbench`, it checks to see if the new component is a circuit component. If it is, then the listener assigns node numbers to the component's terminals and calls the `Solver` object's `addComponent` method. Similarly, a `linkAdded` event results in a call to `addNodes` and a `linkRemoved` or `componentRemoved` event results in a call to `removeNodes`. All of these events, along with `componentEdited` events, result in a call to `findLoops`. This gives the simulation “live” behavior, since any relevant change to the structure of the circuit or to the configuration of any component produces an immediate effect.

Each of the circuit components inherits from an `AbstractCircuitComponent` superclass. This class implements the `WorkbenchComponent` interface, allowing the components to be used directly on the Sieve workspace. This differs from our visualization components, each instance of which is contained in an instance of the generic, default `WorkbenchComponent` implementation.

Circuit components are created by an instance of the `CircuitComponentTool` class. Unlike Sieve's default `Tool` implementation, this class implements simple drag-and-drop functionality. When a `CircuitComponentTool` detects that the mouse has been moved onto the workspace, it asks the new component for its visual representation (by calling the component's `getView` method). The visual representation is then added to the workspace and re-positioned as the user moves the mouse. When the user clicks the mouse the component is “dropped,” and the `Workbench` is notified that a new component has been added.

The circuit components can have multiple terminals, which links can be started from and terminated to. Unlike the links between visualization components, these links — instances of the `CircuitComponentLink` class — do not actually inform the source and destination components of the connection. Links therefore do not represent an acquaintance between the connected components, but instead simply serve as placeholders. Processing of a new link is performed by the `Solver` object as described above.

This application illustrates Sieve's usefulness as a foundation for constructing a collaborative application's user interface. The heart of the application — the `Solver` object — is not dependent on

Sieve. The Sieve-specific implementations of resistor, battery, and other circuit component objects simply provide a way of driving the `Solver`, allowing the simulation to be used collaboratively.

This design works well because the events generated by the `Workbench` translate well into appropriate semantic events for this kind of simulation. We believe that other applications that are similarly modeled on the actions of adding, removing, editing, and linking components could also benefit from a Sieve-based design.

7.3 BeanBox Emulation Design

Since Sieve began as a set of modifications to a JavaBeans-compliant builder tool – the `BeanBox` – it seems natural that we should be able to support collaborative composition of Beans using the connection mechanisms (property binding and events) outlined in the JavaBeans specification. In this section we describe the design of pair of Sieve components that provide this functionality.

Sieve’s default `Tool` and `WorkbenchComponent` components provide part of the original functionality of the `BeanBox`. They allow arbitrary objects to be dropped onto the workspace, and provide access to property editors and customizers. However, the mechanisms used by the `BeanBox` to link components have several characteristics that make them undesirable for direct reuse in a replicated collaborative environment like Sieve. In particular, `BeanBox`’s use of dynamic adapter generation and lack of support for visible links are problematic.

In `BeanBox`, connecting two beans results in the creation of an adapter class that is specific to the source and destination of the connection. Java source code for the adapter is generated, compiled, and loaded to produce the connection. In our collaborative environment, this approach would require generation and compilation of these classes in each collaborating session. A potential problem with this is that it assumes the presence of a compiler in each session’s virtual machine. This may not be a reasonable assumption if, for example, a Sieve session is being executed in a runtime environment such as Sun’s JRE (as opposed to a development environment, such as the JDK). Workspace persistence presents another problem, as the adapter classes may have to be regenerated and re-compiled whenever an existing session is joined.

Connections in the `BeanBox` are also invisible. It is not possible to determine what connections are present simply by looking at the `BeanBox` workspace. This is acceptable if we are simply “taking a snapshot” of the workspace to generate an applet or application. If, however, we are using a builder tool to explore a design collaboratively, then this is undesirable.

Simply making these links visible (e.g., by drawing a line between the connected components) only solves part of the problem, since each link in the `BeanBox` also has state information associated with it. If the link represents property binding, then it would also be useful to show which properties were bound. If the link represents an event notification connection, then it would be useful to show which event invokes which method.

Figure 7.11 illustrates the issues with invisible links. In this example, the “Start Juggler” button has been linked to the juggler animation using `BeanBox`’s event notification mechanism. There is, however, no way to determine that the components are connected in this way, or even if they are

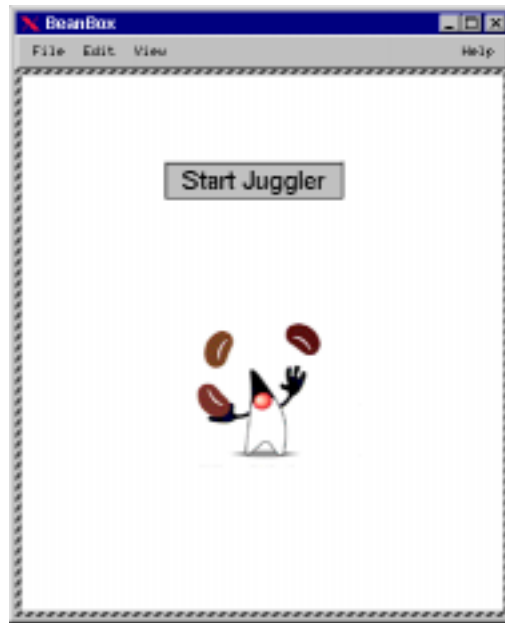


Figure 7.11: A button linked to an animation in the BeanBox.

connected at all. This is appropriate if the tool is being used by a single developer to compose and package the button and animation into an applet, but will cause problems if another person needs to know what connections are present.

Because of the issues with dynamic adapter generation and invisible links, we have implemented generic adapters that use introspection and reflection to connect arbitrary components. These generic adapters – `PropertyBinder` and `EventAdapter` components – provide BeanBox-style property binding and event notification.

An instance of the `PropertyBinder` class takes a pair of objects and a pair of property names, and binds the properties of the two objects. It does this by registering itself as a property change listener in the source object. When notified of a property change, it checks the name of the property to see if it matches the property that it is responsible for binding. If it does match, the `PropertyBinder` gets the new value of the property in the source object and attempts to propagate that value to the appropriate property in the destination object.

Instances of the `EventAdapter` class provide functionality equivalent to the BeanBox’s event propagation mechanism. An `EventAdapter` object, like a `PropertyBinder` object, accepts source and destination objects. It has properties that specify the name of an event generated by the source and the name of a method to be invoked in the destination. When the event property is set, the `EventAdapter` object uses the “encapsulated event” functionality provided as sample code in the BeanBox distribution to create a listener for that event that can be registered with the source. When the specified event is detected, the specified method in the destination is invoked. Currently, our `EventAdapter` implementation supports invoking methods in the destination object that are

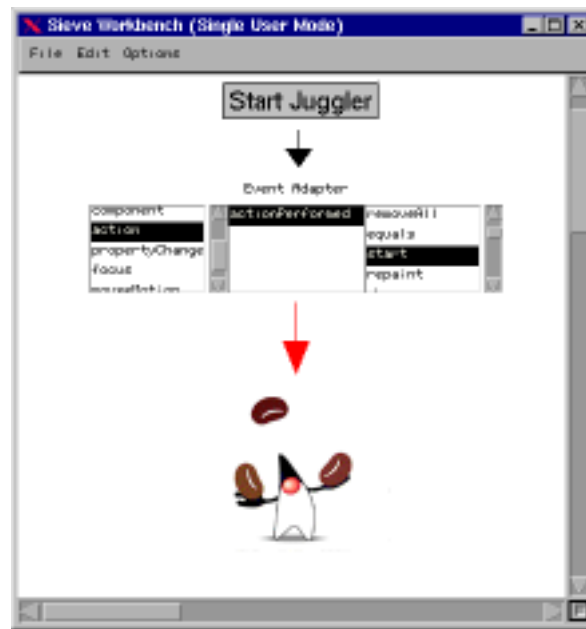


Figure 7.12: A button linked to an animation in Sieve.

either parameterless, or have a single parameter that is assignable from the class of the source object’s event.

The encapsulated event package allows us to avoid using the compiler to dynamically generate specialized listener classes. Intended as a debugging tool, the package includes a specialized class loader implementation that can generate Java byte code for listeners.

Figure 7.12 shows how the same beans used in Figure 7.11 would be connected in Sieve using the **EventAdapter** component. The BeanBox representation is more compact, but the Sieve representation provides more information on how the components are connected. Specifically, the user can see that an **actionPerformed** event (from the button) will result in the animation’s **start** method being invoked.

While both the **PropertyBinder** and **EventAdapter** implementations are usable in runtime environments, there is a speed penalty to our compiler-less approach. The adapter classes generated by BeanBox for property binding or event propagation are very lightweight. They contain only the minimal code necessary to connect two specific objects. Our generalized connector objects must handle a nearly infinite number of cases, since they may connect any two types of objects. They are therefore larger and slower.

We believe, however, that the nature of these objects’ use in Sieve justifies this penalty: A network of beans that is being used for collaborative design or brainstorming would often not need to execute at the same speed as a “released” version of such a network. Once the design activities are finished, the resulting network can be reconstructed in BeanBox (or a similar tool) to produce

BeanBoxComponent
<p>Purpose</p> <p>The <code>BeanBoxComponent</code> interface is implemented by our <code>BeanBox</code> emulation components. It provides an abstraction of an object that is linked between a source object and a destination object.</p>
<p>Methods</p> <pre>public void setSource(Object source) public void setDest(Object dest)</pre>

Figure 7.13: The `BeanBoxComponent` interface.

a more efficient representation with the same functionality.¹ It would also be possible to extend the functionality of the `PropertyBinder` and `EventAdapter` classes to allow them to generate and compile adapter code under appropriate circumstances.

Using components (rather than links) to provide property binding and event notification also makes Sieve’s user interface quite different from that of `BeanBox`, even though the basic functionality is preserved. In an initial design we considered implementing these operations in `Link` objects. This would allow us to make the connections visible on the workspace, but would not provide an obvious way to support changing the configuration of the connections. For example, changing a pair of bound properties would require removing and reconstructing the connection. (This is, incidentally, true in `BeanBox` as well.)

We have therefore implemented `PropertyBinder` and `EventAdapter` simply as objects that may be placed on the workspace using in our default `Tool` and `WorkbenchComponent` implementations. To provide an abstraction for these and any similar components that we may develop in the future, both `PropertyBinder` and `EventAdapter` implement the `BeanBoxComponent` interface, shown in Figure 7.13. This interface requires that an object accept source and destination objects. Customizers are provided for both the `PropertyBinder` and `EventAdapter` that help ensure that, for example, the user can only select compatible properties for binding.

`BeanBoxComponent` instances may be linked to any component on the workspace using a custom `Link` implementation (the `BeanBoxLink`), which we register with the `LinkFactory`.

Specifically, we register this link twice, as shown in Figure 7.14. This tells Sieve to use a `BeanBoxLink` instance to connect *any* object to or from a `BeanBoxComponent` instance. Although registration may be done anywhere, we currently call the methods shown in Figure 7.14 during static initialization of the `EventAdapter` and `PropertyBinder` classes.

¹Assuming, of course, that only `PropertyBinder` and `EventAdapter` components, along with non-Sieve-specific beans, were used. Reproducing Sieve-specific entities such as the links between dataflow components would not be possible in `BeanBox`, but could be accomplished in more advanced builder tools by hand-coding the connections.

```
LinkFactory.getLinkFactory().registerLink(BeanBoxComponent.class,  
    Object.class, BeanBoxLink.class);  
  
LinkFactory.getLinkFactory().registerLink(Object.class,  
    BeanBoxComponent.class, BeanBoxLink.class);
```

Figure 7.14: Registering `BeanBoxLink` with the `LinkFactory`.

Figure 7.15 shows the implementation of the `BeanBoxLink` class with error checking and exception handling removed. The `BeanBoxLink` can assume that either its source or destination component is a `BeanBoxComponent`. If the source is a `BeanBoxComponent`, then that object's `setDest` method is called, passing the destination object as a parameter. If the destination is a `BeanBoxComponent`, then that object's `setSource` method is called, passing the source object as a parameter. When the link is broken, the `BeanBoxComponent`'s source or destination is set to null.

```
public class BeanBoxLink extends SimpleLink {

    // The BeanBoxComponent involved in this link
    private BeanBoxComponent bbc = null;

    // True if our destination object is a BeanBoxComponent
    private boolean bbcIsDest = false;

    public void link(WorkbenchComponent s, WorkbenchComponent d) {
        // Get the objects that the two WorkbenchComponents represent
        Object src = s.getModel();
        Object dest = d.getModel();

        // Figure out which direction we are linking
        if (src instanceof BeanBoxComponent) {
            BeanBoxComponent bbc = (BeanBoxComponent)src;
            bbc.setDest(dest);
            bbcIsDest = false; // Remember which direction we are going
        }
        else if (dest instanceof BeanBoxComponent) {
            bbc = (BeanBoxComponent)dest;
            bbc.setSource(src);
            bbcIsDest = true; // Remember which direction we are going
        }
    }

    public void unlink(WorkbenchComponent s, WorkbenchComponent d) {
        // On unlink, just set the source or dest to null
        if (bbcIsDest)
            bbc.setSource(null);
        else
            bbc.setDest(null);
    }
}
```

Figure 7.15: The BeanBoxLink implementation, with error checking removed.

Chapter 8

Conclusions

We conclude by summarizing the benefits of collaborative component workspaces, discussing the contributions of the Sieve design, and suggesting directions for further investigation.

8.1 Benefits of a JavaBeans-Based Collaborative Workspace

The applications that we have presented demonstrate the power of a collaborative workspace as a tool for cooperative design. Building on the benefits of shared whiteboard tools, shared workspaces that allow manipulation and composition of arbitrary active components present interesting new opportunities.

Basing the workspace design on a popular component architecture such as JavaBeans should greatly increase the probability that useful, externally-developed components can be brought into the workspace environment. Even if such components cannot be used directly, the emphasis of component architectures on complete and consistent interfaces suggests that the task of adapting third-party components will likely be much simpler than rebuilding the components' functionality from scratch.

Finally, choosing Java as an implementation platform simplifies distribution of the software and largely eliminates platform and operating system compatibility issues. While these characteristics of the Java language are useful for software in general, they are of critical importance for collaborative software: Users whose computers cannot communicate, either because of lack of software or platform incompatibilities, cannot take advantage of computer-supported collaboration.

8.2 Contributions of the Sieve Design

We have presented the design of the Sieve framework, along with our preliminary set of application components for visualization, circuit simulation, and software construction. These components demonstrate the feasibility of a collaborative component workspace design.

The underlying themes to our objectives in designing Sieve are flexibility and extensibility. We provide reasonable defaults that allow many arbitrary components to be manipulated collaboratively in useful ways. Where these defaults are not reasonable, we provide well-defined ways of extending and overriding basic functionality. Developers can add new ways to create, display, edit, and connect components. Components may use Sieve's default state replication mechanism simply by following basic JavaBeans conventions. Where this is not possible or appropriate, developers may implement custom mechanisms for replicating component state. Hence collaboration-unaware components are supported by default, but developers are free to construct or adapt collaboration-aware components when necessary.

All of these extensions can apply to groups of components (e.g., based on a common subclass or interface), or to individual component classes. Custom Sieve-specific components may also be written if the developer desires the highest possible level of control over the interaction between a component and the workspace environment.

Sieve's design addresses many common collaboration support issues. It supports location awareness through telepointers and a radar view. The radar view permits location-relaxed WYSIWIS, enhancing flexibility by allowing users in the same workspace to either work together or in isolation. Server-based persistence allows users to join a session already in progress, and also supports asynchronous collaboration.

Finally, the modular structure of Sieve's core components and collaboration support mechanism allows developers to use the workspace functionality in a variety of ways. Instead of a monolithic application, we provide a set of reusable classes that can be composed into an applet, a stand-alone application, or integrated with a larger piece of software.

8.3 Future Work

As we have explored new ways of using Sieve and written new application components, we have discovered a number of opportunities for further development. This section outlines possible changes and extensions to core Sieve components and to Sieve's collaboration support mechanism.

8.3.1 Extensions to Sieve Workspace Functionality

A number of possible extensions or enhancements to the behavior of the core Sieve components have been proposed.

Many aspects of the Sieve's user interface need further study and improvement. Typically, these enhancements would be made at the level of individual `WorkbenchComponent` implementations. For example, many users have expressed a dislike for the way in which links are started from the default `WorkbenchComponent` implementation. Components that use this implementation (including all of our visualization components) are linked by right-clicking on the component's border, moving the mouse to the border of a destination component, and then clicking on the destination component's border to terminate the link. Further study is needed to determine an appropriate replacement for this interface mechanism.

Currently, `Tool` objects construct new components by listening for mouse events generated by the `Workbench`. It might, however, be useful to generalize this to allow construction from other types of events. For example, a `Tool` implementation could listen for both mouse events and key events generated by user actions on the workspace. The `Tool` could then decide how to construct a new component based not only on what the user did with the mouse, but also by what they typed.

Under the current model, only a single component may be selected at a time. Extending `Sieve` to allow multiple component selections would make many workspace operations less tedious, but would require careful reconsideration of other behavior currently associated with selection. The current framework support for editing and linking, in particular, would need to be revisited.

If multiple selection functionality was available, grouping of multiple components into a single manipulable unit would be a logical extension. This is analogous to the functionality of many object-based drawing programs, which allow a group of objects to be grouped. The entire group is then treated as a single object that can be moved, resized, deleted, or included as part of a larger group.

Of the eight objectives outlined in Chapter 4, support for aggregate components is probably the least thoroughly achieved in our current design. A developer can write a component whose sub-components may be individually edited or linked, but doing so requires considerable effort. Future versions of the JavaBeans architecture will likely have improved support for describing aggregate components. An early draft of the “Glasgow” specification [35] (JavaSoft’s description of near-future JavaBeans enhancements) included a proposal for an “Object Aggregation/Delegation Model For Java and JavaBeans.” Under this design, any object could implement a standard `Aggregate` interface and present a set of interfaces that it supported. Clients of such an object could then request an instance of any of these interfaces. If this interface (or one similar to it) is adopted and added to the JavaBeans specification, `Sieve` should use it to provide additional support for aggregate components.

Future work on support for aggregate components should account for extensions made to support multiple selection and grouping. If support is added for grouping components on the workspace into a single component, this group should be manipulable in the same way that hand-coded aggregates are. If standards for aggregate beans do emerge, then a set of grouped components should conform to this standard. `Sieve` can then simply support this standard, and treat these types of grouped components the same way that it treats hand-coded aggregates.

Our current distributions of `Sieve` allow the workspace to be viewed by starting a stand-alone application rather than, for example, an applet. This eliminates one of the primary advantages of Java, the ability to transparently download software embedded in web pages. Fortunately, this is not an intrinsic limitation of our implementation. We have taken care to, as much as possible, avoid using functionality in core `Sieve` components that may not be available when running in a browser-based virtual machine. At least two potential problem areas remain:

- Popular Java 1.1-compliant web browsers do not allow applets to use the JavaBeans introspection mechanism fully. Occurs when the introspector attempts to obtain a list of all of the names of methods in a class. Currently this returns information on public, protected, and private methods. Since, however, reflective access to private methods is restricted in applets,

a security exception is thrown and introspection fails. Explicitly describing component classes using standard JavaBeans BeanInfo files and using a modified introspection mechanism may provide a solution.

- Applets are not allowed to use custom class loaders. This, for example, prevents an applet from installing a class loader to import a Java archive (JAR) file with a set of components. This is largely a packaging problem, and can likely be avoided by ensuring that all component classes that the user will need to add to the workspace are loadable from the same location as the applet classes.

The security mechanisms used by browsers to prevent malicious Java applets from damaging local data are constantly evolving. It may at some point be possible to eliminate the restrictions on a Sieve applet by digitally signing the applet archive, or by instructing the browser to give the applet special privileges in some other way.

Finally, Sieve's error-handling and debugging functionality could be significantly enhanced. The Sieve framework is exposed to potentially fatal errors in any component added to the workspace. Where these errors can be detected by Sieve code, it may be possible to simply remove the offending component and continue. A more interesting approach would allow expert users to explore the problem and perhaps re-load a revised version of the broken component. (Kansas [28] provides this type of functionality.)

8.3.2 Extensions to Sieve Collaborative Functionality

Sieve's implementation currently assumes that all sessions have access to the same class definition files. For example, we currently attempt to load all classes via the virtual machine's default classloader. If a given class cannot be found, we check all JAR files loaded by a custom classloader. If one Sieve user instantiates a class that a remote user does not have access to, the instantiation will fail on the remote machine. A number of solutions to this problem are worthy of further investigation. The Sieve instance that first creates a component could either send the class file to all other sessions, or allow the other sessions to retrieve the class file on demand. The set of available components could also be restricted to components available on the server, with means provided for users to upload new component sets (e.g., in JAR files) to the server.

The Sieve server has no true user interface. Limited command-line configuration options are provided, but there is, for example, no provision for monitoring server activity. Maintenance tasks such as removing the persistent record of an unneeded session must also be done from the command line. Creating an administration client, perhaps as a component that could be included in an arbitrary applet or application, would significantly enhance the server's usability.

Additional investigation into the use of alternate persistence mechanisms, likely based on Java's Object Serialization, could perhaps produce a design superior to our event replay mechanism for supporting late-joiners and saving the workspace state. Any such investigation should also explore extensions to support saving local copies of a workspace configuration.

Bibliography

- [1] G. Abram and L. A. Treinish. An Extended Data-Flow Architecture for Data Analysis and Visualization. In *IEEE Visualization '95*, pages 263–270. IEEE, October 1995.
- [2] E. Alard and G. Bernard. Preemptive Process Migration in Networks of UNIX Workstations. In *Proceedings of the 7th International Symposium on Computer and Information Sciences*, pages 129–140, 1992.
- [3] Niam Alper and Chuck Stein. Geospatial metadata querying and visualization on the www using Java (tm) applets. In *Proceedings of Visualization '96*, pages 77–84. IEEE, October 1996.
- [4] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, 1996.
- [5] James Begole, Craig A. Struble, and Clifford A. Shaffer. Leveraging Java Applets: Toward Collaboration Transparency in Java. *IEEE Internet Computing*, 1(2):57–64, March-April 1997.
- [6] James ‘Bo’ Begole, Craig A. Struble, Clifford A. Shaffer, and Randall B. Smith. Transparent Sharing of Java Applets: A Replicated Approach. In *1997 Conference on User Interface Software and Technology (UIST'97)*, October 1997.
- [7] CoVis: Learning Through Collaborative Visualization. URL: <http://www.covis.nwu.edu/>.
- [8] T. Crowley, E. Baker, H. Forsdick, P. Milazzo, and R. Tomlinson. MMConf: An infrastructure for building shared applications. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '90)*, Los Angeles, California, 1990. ACM Press.
- [9] D. Scott Dyer. A dataflow toolkit for visualization. *IEEE Computer Graphics and Applications*, 10(4):60–69, July 1990.
- [10] Colin Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1), February 1988.
- [11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [12] S. Greenberg, M. Roseman, D. Webster, and R. Bohnet. Issues and experiences designing and implementing two group drawing tools. In *Proceedings of Hawaii International Conference on System Sciences, 4*, pages 138–150. IEEE Press, 1992.
- [13] GroupKit Home Page. URL: <http://www.cpsc.ucalgary.ca/projects/grouplab/groupkit/>.

- [14] Introduction to the General Social Survey. URL: <http://www.icpsr.umich.edu/GSS/about-gss/gssintro.htm>, 1996.
- [15] C. Gutwin, S. Greenberg, and M. Roseman. A Usability Study of Awareness Widgets in a Shared Workspace Groupware System. In *Computer-Supported Cooperative Work*, pages 258–67. ACM Press, 1996.
- [16] B. Henderson-Sellers. OPEN Relationships — Compositions and Containments. *Journal of Object-Oriented Programming*, pages 51–55, November/December 1997.
- [17] Hewlett Packard: HP’s World Class X Server - Features - SharedX. URL: <http://www.hp.com/xwindow/viaFrames/features/sharedx.html>.
- [18] Ralph D. Hill. Languages for the construction of multi-user multi-media synchronous (MUMMS) applications, 1992.
- [19] G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.
- [20] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [21] J. Chris Lauwers, Thomas A. Joseph, Keith A. Lantz, and Allyn L. Romanow. Replicated architectures for shared window systems: A critique. In *OIS90, Computer Mediated Work Environments*, pages 249–260. 1990.
- [22] J. Chris Lauwers and Keith A. Lantz. Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. In *Proceedings of ACM CHI’90 Conference on Human Factors in Computing Systems*, pages 303–311, 1990.
- [23] Microsoft NetMeeting. URL: http://www.microsoft.com/products/prodref/113_ov.htm.
- [24] Penumbra Software Super Mojo. URL: <http://www.penumbraoftware.com>.
- [25] Constantinos Phanouriou and Marc Abrams. Transforming command-line driven systems to web applications. In *Proceedings of the Sixth International World Wide Web Conference*, April 1997.
- [26] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, Wokingham, 1995.
- [27] Rational Software Corporation UML Resource Center. URL: <http://www.rational.com/uml/>.
- [28] Randall B. Smith, Mario Wolczko, and David Ungar. From Kansas to Oz: Collaborative debugging when a shared world breaks. *Communications of the ACM*, 40(4):72–78, April 1997.
- [29] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS Revised: Early Experiences with Multiuser Interfaces. In *ACM Transactions on Office Information Systems*, pages 147–167. ACM Press, April 1987.

- [30] Java Shared Data API. URL: <http://java.sun.com/people/richb/jsda>, 1996.
- [31] JavaBeans (tm). URL: <http://splash.javasoft.com/beans/beans.100A.pdf>, December 1996.
- [32] HotJava (TM) HTML Component. URL: <http://java.sun.com/products/hotjava/bean/>, 1997.
- [33] InfoBus Specification. URL: <http://java.sun.com/beans/infobus/ibspec.html>, 1997.
- [34] Java AWT: Delegation Event Model. URL: <http://java.sun.com/products/jdk/1.1/docs/guide/awt/designspec/events.html>, 1997.
- [35] JavaBeans Activation Framework (JAF). URL: <http://www.javasoft.com/beans/glasgow/jaf.html>, 1997.
- [36] JavaSpace Specification. URL: <http://chatsubo.javasoft.com/javaspaces/js-spec/index.html>, 1997.
- [37] Swing: The Preliminary Specification. URL: <http://java.sun.com/products/jfc/swingdoc-current/doc/index.html>, 1997.
- [38] J. C. Tang. Findings from observational studies of collaborative work. In *International Journal of Man Machine Studies*, volume 34, pages 143–160. special edition, 1991.
- [39] Craig Upson, Thomas A. Faulhaber, Jr., David Kamins, David Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The Application Visualization System: a Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.
- [40] Visual Numerics JWAVE. URL: <http://www.vni.com>.
- [41] Jason D. Wood, Ken W. Brodlie, and Helen Wright. Visualization over the world wide web and its application to environmental data. In *Proceedings of Visualization '96*, pages 81–86. IEEE, October 1996.

Appendix A

UML Overview

Throughout our design descriptions we use a subset of Rational Software’s Unified Modeling Language (UML) [27]. UML is a language for specifying, visualizing, constructing, and documenting software artifacts. In this appendix we provide a brief overview of the parts of the UML notation that we use.

A.1 Class Diagrams

Figure A.1 shows a UML class diagram. Rectangles represent concrete classes and rounded rectangles represent pure abstract classes, or “interfaces” in Java terminology. Since the diagram is just providing an overview of this set of classes, attributes and methods of the classes are not shown.

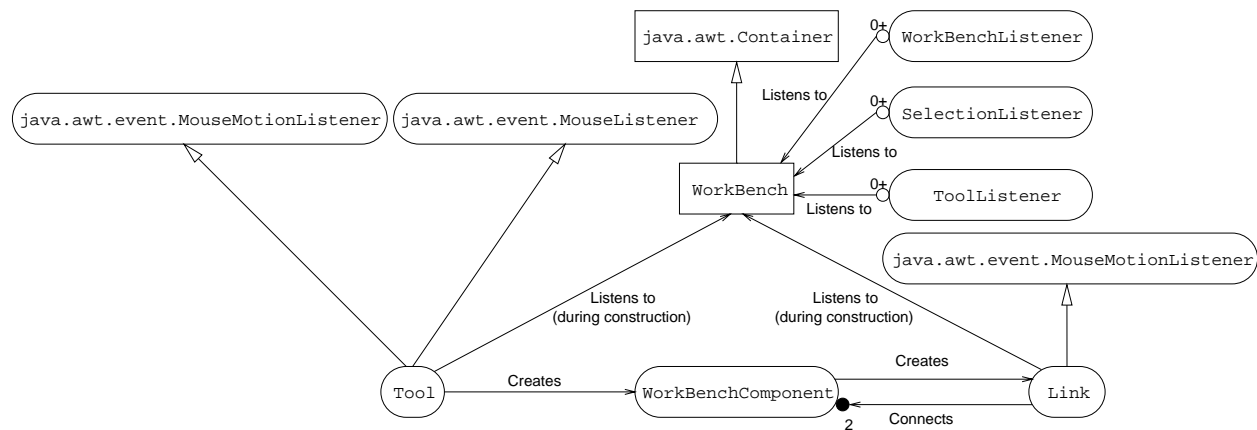


Figure A.1: UML class diagram.

The diagram illustrates both the static inheritance structure of the set of classes and the dynamic associations between those classes. Inheritance is indicated by hollow arrows pointing from a subclass to its superclass. The `Workbench` class, for example, inherits from the `java.awt.Container`

class. Similarly, the `Link` interface inherits from (or “extends,” in Java terminology) the `java.awt.event.MouseMotionListener` interface.

Dynamic relationships (or “collaborations”) are indicated by arrows labeled with the name of the relationship. For example, `Tool` objects create `WorkbenchComponent` objects. Dynamic relationships that are not one-to-one are indicated by circles at one end, labeled with the order of the relationship. A hollow circle represents a weak relationship, such as “uses” or “knows of” association. For example, zero or more `WorkbenchListener` objects may know of and listen to a single `Workbench` object. A filled circle represents a stronger relationship, such as “contains,” “controls,” or “owns.” A `Link` object, for example, controls two `WorkbenchComponent` objects by connecting them in some way.

When using these diagrams to describe the design of Sieve, we attempt to focus on what we believe are the critical relationships between the classes in our design. For example, the fact that `Workbench` extends `java.awt.Container` is important, since it allows a `Workbench` instance to contain the visual representation of Sieve components. The diagram does not, however, show that `java.awt.Container` inherits from `java.awt.Component` or that `java.awt.Component` inherits from `java.lang.Object`. These relationships are necessary for the implementation to actually work, but are not as relevant to the discussion of Sieve’s design.

A.2 Interaction Diagrams

Class diagrams like the one shown in the previous section can be used to illustrate associations at a high level. Lower-level interactions between objects can be described using UML’s interaction diagrams.

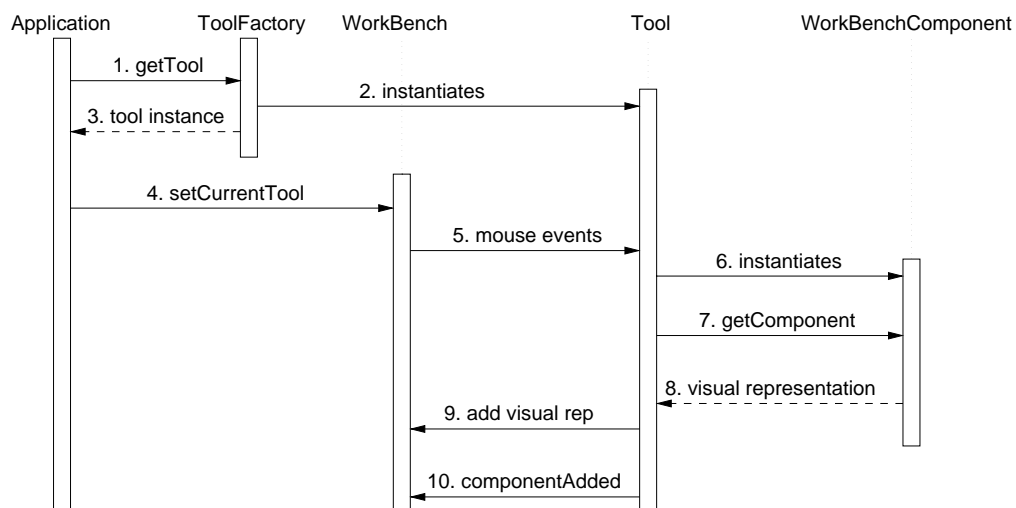


Figure A.2: UML interaction diagram.

Figure A.2 shows an interaction diagram. The names of the objects involved in this interaction are shown at the top of the diagram. The “messages” passed between objects are indicated by labeled arrows. The scenario described by this diagram could be narrated as follows:

The application invokes `getTool` on the `ToolFactory`. The `ToolFactory` instantiates an appropriate tool implementation and passes it back to the application. The application then tells the `Workbench` to set the new `Tool` as its current tool. The `Workbench` will then forward any mouse events to the `Tool`. The `Tool` will instantiate a `WorkbenchComponent` based on these events. The `Tool` will then ask the new `WorkbenchComponent` for its visual representation, and add it to the `Workbench`. Finally, the `Tool` will tell the `Workbench` that a new component has been added.

As with class diagrams, we use interaction diagrams to illustrate ideas which are critical to Sieve’s design. This means that the code which implements a given sequence of object interactions will typically contain many more method invocations than the number of interactions shown in a diagram like the one in Figure A.2.

Vita

Philip Locke Isenhour was born in Hickory, North Carolina on June 8, 1973. He began his undergraduate studies at Virginia Tech in August, 1991. As an undergraduate he participated in the cooperative education program, working for five semesters at Bell Northern Research (now Nortel) in Research Triangle Park, North Carolina.

He received his Bachelor of Science degree in Computer Science from Virginia Tech in May, 1996 and remained at Virginia Tech for his graduate studies. He will be graduating with a Master of Science degree in Computer Science in May, 1998.