Sluice: A Java-Based Framework for Collaborative Interactive Modular Visualization Environments

Philip Isenhour Clifford A. Shaffer James "Bo" Begole Jeff Nielsen
Marc Abrams
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061
{isenhour, shaffer, begolej, jnielsen, abrams}@cs.vt.edu

Abstract

We present **Sluice**, a framework for constructing collaborative, interactive modular visualization environments (CIMVEs) for use on the World Wide Web. **Sluice** supports using Java to build modules that create, manipulate, and generate visualizations of tables of data. **Sluice** provides necessary infrastructure for a CIMVE, including interface specifications for methods that modules of the MVE must implement, event propagation to support interaction and dynamic updating, and mechanisms for efficiently supporting collaboration. Our fundamental description of data comes in the form of table which is filtered, extended, or otherwise manipulated to produce a new "view" of the data at each module.

We use the observer/observable design pattern to support dynamic updating of visualizations downstream in the dataflow network. **Sluice** also supports on-the-fly manipulation of the dataflow network. **Sluice** uses functionality provided by the JavaBeans package to support introspection of modules. This allows CIMVE implementations to discover, query, and set the values of properties of modules at runtime.

After describing the design rationale, capabilities, and implementation of **Sluice**, we then describe our prototype CIMVE named **Sieve**.

1 Introduction

In recent years, Modular Visualization Environments (MVEs) such as AVS [14], Data Explorer [1], and apE [6] have received a great deal of attention. These systems present the user with an extensible 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. The success of these and similar packages has demonstrated the power of direct manipulation of dataflow networks as a user interface mechanism.

At the same time, the explosion in popularity of the World Wide Web has opened opportunities for distributing both raw data and visualization tools to a wider user population. Systems have been developed which provide a Web-based forms interface to drive visualization software on a remote server [17]. 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.

The emergence of Sun's Java programming language [3] has further expanded the potential of the Web as a delivery mechanism for remote data and tools. Java programs known as "applets" may be embedded in Web pages to, among other uses, provide an interactive interface to data and software on remote servers. Alper, et al. [2] describe an applet which allows exploration of geospatial datasets. Data is 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 which generates visualizations remotely [15]. In this case the applet is used to drive the remote software and display the resulting image. The use of interactive applets represents a significant step forward from the batch-mode orientation of forms-based interfaces. 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 MVEs.

The Web also provides infrastructure for building collaborative systems. Asynchronous collaboration methods such as e-mail and Web posting have become quite common, but relatively little support exists for real-time, synchronous collaboration. In this paper we present the argument that providing support for synchronous collaboration in visualization environments greatly increases the functionality of these systems.

One advantage of MVEs is the ability to easily alter the data flow network to search for specific visualizations that illustrate useful characteristics of a dataset. This capability can be best leveraged if the environment provides for interactive manipulation of the dataflow network: Where resources permit, an MVE should support dynamic updating of visualizations based on changes to the configuration of the dataflow network. The feedback provided by this type of interaction facilitates exploration in a way that systems which impose a distinction between modes of network design and network execution cannot.

In this paper we give an overview of **Sluice**, a framework for constructing collaborative interactive modular visualization environments (CIMVEs) in Java. **Sluice** provides support for linking separate computation and visualization modules to create dataflow networks. Source modules produce or read data and present it as a two-dimensional table containing objects of any type supported by the Java language. Processing modules can manipulate this data and present an altered or extended table. Visualization modules can then produce images of the data in a table, and also serve as an interface for data selection. The resulting dataflow network can be made fully interactive, using an event mechanism to notify downstream network modules of changes to the data or to the configuration of the network. Downstream modules can then retrieve new or modified data from the source. **Sluice**-based systems can be delivered through the Web, and they use the Web for data retrieval and for collaboration. Support for collaboration is provided by an efficient mechanism for propagating changes to the configuration of a dataflow network in such a way that modules need not be explicitly programmed for collaboration.

1.1 Real-time Collaborative Analysis

People interact together in all aspects of life and, as computers have become prevalent, people seek computer support for these interactions. Collaborations range from asynchronous, where the points of interaction are separated by relatively long periods of time, to synchronous, where the interactions are separated by short periods of time or even take place simultaneously. Asynchronous collaborations are easy to support with current technology. Files may be posted on the Web for broad public distribution, or attached to email messages for narrow person-to-person or small group sharing. However, there is little support for real-time collaborations. There are three general situations where data analysts can work together concurrently.

Presentation. For others to benefit from the results of an analysis, the analyst must present them. To do so with present systems, the analyst needs to package the demonstration in a distributable format, such as a print-out or standard electronic file format. The analyst may attempt to anticipate questions about the presentation and prepare answers. Unfortunately, it is impossible to anticipate all questions. Alternatively, if the presenter and audience share simultaneous access to the material, the analyst can respond to questions in real-time, perhaps generating new views which more clearly present an aspect of the data that was previously unclear. Furthermore, with the ability to manipulate the data directly, the audience may use the visualization to illustrate their questions and misunderstandings. The discussion is *grounded* by the shared view [5].

Interpretation. Colleagues commonly assist each other when interpreting data. However, few software environments exist that allow more than one person at a time to interact with the system. To collaborate with current visualization tools, an individual runs the input devices of a single machine while other participants "back-seat drive" the session. Switching drivers quickly becomes infeasible as the number of participants grows.

Roles. People work together to achieve a common goal. However, collaborators may not share the same sub-goals since their specific roles in the collaboration may differ. This is true in visualization as in any activity. For example, imagine two or more analysts viewing real-time weather data of a storm. One may be interested in determining the *path* of the storm and the other its *intensity*. Sharing the same data set, these two may assist each others' interpretations on the way to achieving their individual ends.

2 Sluice Design Overview

Our goal in designing **Sluice** was to provide support for using Java to build modules that create, manipulate, and generate visualizations of tables of data. These modules can then be used in a variety of applications, with our primary emphasis being on constructing CIMVEs. **Sluice** is not, however, an implementation of such an environment. We discuss our prototype **Sluice**-based CIMVE, **Sieve**, in Section 4.

The **Sluice** framework can be broken into three distinct parts:

- A set of **interface specifications** which define methods that **Sluice** modules must implement. The `TableView` interface, discussed in more detail below, represents the minimum requirements for a **Sluice** component: Any class which implements this interface can be used (at least in a minimal way) in a **Sluice**-based system.
- A set of **design patterns** which define conventions for names and method signatures. Conformance to these patterns increases a component's flexibility when used in **Sluice**-based systems.
- A set of **support classes** which implement commonly-needed functionality such as event propagation. These classes are discussed in Section 2.2.1.

2.1 Defining Sources of Data

Sluice uses the abstraction of a simple two-dimensional table of cells as the fundamental structure for data manipulation. These tables may be of any size and may contain objects of any type supported by Java, including aggregate types such as vectors or arrays.

Subclasses of an abstract `Table` class encapsulate the details of interaction with a specific type of data source to create the initial table in a dataflow network. For file or Web-based data, these objects will typically be parsers, converting raw data streams into a table of Java objects. For data which is taken from a remote database, `Table` objects hide the details of query and access mechanisms specific to those sources. For example, `Table` implementations using the Java Database Connectivity (JDBC) package [11] can access SQL databases, and `Table` implementations using Java Remote Method Invocation (RMI) [10] or JavaIDL [12] can be used to access data on Java-based or CORBA-compliant

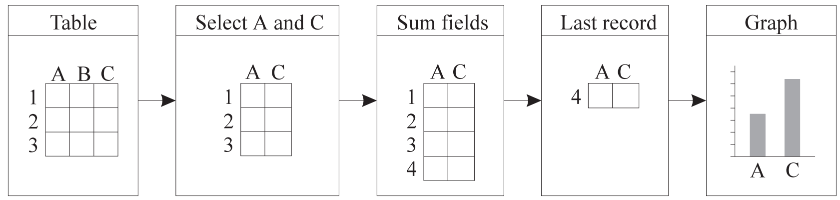


Figure 1: A series of manipulations on a table of data. Each manipulation produces a new "view" of the table.

servers, respectively. A simulation applets or other Java-implemented data producer may also provide a `Table` implementation to access its output.

In our current implementation, sources of data are specified by URL-like strings which encode both the format and location of the raw data. **Sluice** provides a single object which parses this string and returns an instance of the specific `Table` subclass which understands the type of data provided by the source. Among other advantages, the use of a factory mechanism allows new `Table` implementations to be added to the system easily. In fact, Java's dynamic class loading capabilities allow these implementations to be added "on the fly," without having to restart the applet or application. Eventually, we would like to provide the ability for `Table` objects to test a given data source to see if they understand it. The use of custom MIME-types may also provide a mechanism for hiding data format specification from the user [17].

`Table` objects, having specific knowledge of raw data formats, may also be able to provide "hinting" information for downstream components to examine. For example, specific types of `Table` objects could provide knowledge of which fields represent independent variables, label strings, or other interesting types of data. Hints may also suggest such things as a range for a field, such that data falling outside of this range should be treated as outliers. A set of standard hints (identified by string constants) is defined by the system, and applications can also add new ones.

2.2 Representing a "View" of the Data

The state of a dataset in a **Sluice**-based dataflow network is represented by a "view" of a table at each module in the network. The initial view is provided by a `Table` object and consists of a table of the data as it was retrieved from the source. The next module in the network will take the output from the `Table`, manipulate it in some way, and generate an altered view of the data. This view will also be a two-dimensional table, but may be of a different size or contain objects of different type than those in the original view. The next module can then use this view to produce yet another view, with this process continuing until the end of the network is reached.

Figure 1 gives an example of this. The original view of the data is a 3 by 3 table. This view is used as input to a Field Selection view, which selects fields A and C, hiding field B. The next view sums the values in the fields and adds a row. This view is then the source for a view which selects only the last record. The graph then takes this view of the table to produce an image.

Conceptually, each module contains a complete copy of the view of the table and it makes available to the next module in the network. For modules which only alter the structure but not the content of the table, this approach may be inefficient. These modules will often simply restructure queries and propagate them to their source module. For simplicity, we will ignore this behavior in our discussion and describe modules as if they actually stored a copy of their view of the data internally.

In **Sluice**, the concept of taking a view of a table as input and generating another view of a table as output is represented by the `TableView` interface. An interface in Java is similar to an abstract class. Interfaces may define method signatures, but may not contain any implementation code. Java classes may only extend a single superclass, but may also implement an arbitrary number of interfaces. This mechanism achieves a effect similar to multiple inheritance in **C**++.

The specification of `TableView` as an interface rather than as a class allows new classes to be written specifically to act as `TableView` objects, but also allows existing implementations of everything from simple GUI components to entire Java applets to be extended and used in **Sluice**-based environments without requiring changes to the location of these classes in the class hierarchy.

The basic function of all `TableView` objects is to consume something that looks like a table and generate something else that looks like a table. This definition is sufficiently vague as to allow a wide range of functionality to be provided by `TableView` objects. Our current suite of `TableView` objects includes the following kinds of views:

- **Selection Views**, which hide one or more fields of the table. One application of this type of view in an MVE is to simply reduce a table to a more manageable size by hiding fields that are not needed for a given analysis. Another application is to hide fields which are not of a given type.

- **Sample Views**, which sample values from a table based on some sampling mechanism. The output of a sample view is a complete table containing only the sampled values.

- **Translation Views**, which translate values in one or more fields. For example, nominal data is often represented by numeric values, with corresponding labels provided as meta-data. One type of translation view converts these numeric values to the appropriate string labels.

- **Criteria Views**, which hide all records which do not meet a given criteria. Using this type of view, a query on a table creates a new table containing only the values that meet the criteria.

- **Derived Views**, which introduce new fields based on calculations on other fields. Such a view could compute a new field based on an arbitrary formula to emulate the functionality of a spreadsheet. The derived fields added by this type of view may be of any class, so this functionality could, for example, be used to map existing values directly to colors, sounds, or geometric representations.

While many types of `TableView` objects represent manipulations that are interesting to users of MVEs, the functionality of `TableView` objects is often useful when developing applications outside of visual environments. The original `TableView` design was modeled after Java's `FilterInputStream` interface. In Java, an abstract `InputStream` superclass is defined, with subclass implementations providing streams for reading data from files, sockets, and other sources. Common functionality such as buffering or parsing is provided by wrapping implementations of the `FilterInputStream` class around the `InputStream`. While we have extended this concept considerably to allow for dynamic linking of `TableView` objects, all `TableView` implementations can still be used in this way.

Since `TableView` objects used in non-visual coding of applications may have no need for GUI components to manipulate their behaviors, it is advantageous *not* to write `TableView` objects as GUI components, even though visual controls may be needed when the modules are used in an MVE. Enforcing a distinction between these would likely result in two separate libraries of views, one with visual controls built in and one without. In Section 3.1 we present our solution for avoiding this.

`TableView` objects can be linked together in networks of arbitrary size and with arbitrary branching. We anticipate that most `TableView` objects will take a single source, but **Sluice** does not require this. (A Java exception is provided that may be thrown by single-source `TableView` implementations if an attempt is made to connect the object to more than one source.)

2.2.1 Adapting Existing Classes

Simple visualization applets were among the first practical uses of the Java language. Sun's Java Development Kit ships with several such applets for doing simple charts and graphs. Many more are freely available from applet repositories on the Web. While existing applets or class libraries for doing various kinds of filtering or other data manipulation are perhaps somewhat less common, these could also represent a valuable source of re-usable code for MVEs.

To facilitate adaptation of these existing classes for use in **Sluice**-based MVEs, we have attempted to keep the `TableView` interface as simple as possible. To convert any (appropriate) existing class to a `TableView` implementation, it is only necessary to extend the class by adding methods for handling the table input and output, specifically:

1. Getting the names of available fields
2. Querying basic meta-data (type, description, and level) for each field
3. Adding and removing data sources
4. Handling events from upstream modules and propagating them to downstream modules.
5. Getting a chunk of cells based on a list of field names and a range of records

Much of this functionality can be shared by multiple `TableView` implementations, so basic `TableView` superclasses for both applets and non-visual objects are provided. These support classes essentially implement a "transparent" view of their source, allowing subclasses to override appropriate methods to perform the appropriate manipulations. In this way Java's interface mechanism gives us the best of both worlds: We can provide common functionality in the form of support classes, but we need not require that all classes used in **Sluice**-based systems inherit from them. In cases where these superclasses cannot be used, other support classes are provided for operations such as event handling.

2.3 Visualization Components

Sluice does not explicitly distinguish between terminal and processing modules. All components which actually generate visual representations of data may implement the `TableView` interface and may therefore be placed anywhere (downstream of a source module) in a dataflow network.

This does more than just simplify our design. It reflects the fact that visualizations of data often provide intuitive interfaces for selecting further filtering or other manipulations. Simple visualizations such as scatterplots and line graphs lend themselves to data selection techniques such as those outlined by Wills [16]. For example, if a scatter plot reveals outliers in the dataset, these could be selected for further analysis by clicking on them or dragging across them. Downstream modules can then "see" a table containing only those values, or at least distinguish between selected and unselected data points using the hinting mechanism described in Section 2.1. If the outliers are determined to be noise in the data, the interesting values could be selected for further processing in a similar manner.

This functionality could be achieved in other ways, but we hope that the lack of a specific definition for what a terminal component may or may not do will encourage the development of visualizations which also act as filters.

2.4 Supporting Interaction

One of the primary goals of the **Sluice** framework is to support **interactive** visualization, in which changes to parameters are immediately reflected in the visualization. This differs from visualization environments that are essentially **batch**-oriented, where the analyst sets parameters for a visualization before explicitly starting execution of the system. An interactive system, on the other hand, facilitates understanding of relationships among data, because the analyst can immediately see the effect of changes made to the data flow. It may be that the dataset is so large or the computation is so complex that a system cannot provide immediate feedback of changes. Interactive visualization should be supported, however, whenever feasible.

To support real-time interactive visualization, **Sluice** uses the **observer-observable** object-oriented design pattern, described by Gamma et al. [7]. In this model, an **observer** registers itself with an **observable** object, which notifies the observer whenever the observable's state changes. We adopted this approach from the **delegation event model** of the Abstract Window Toolkit (AWT) of Java 1.1.

Whenever the state of a `TableView` changes it generates a `TableViewEvent` and delivers it to any registered listener object. In the most common case, `TableView` objects will register themselves as listeners to their source `TableView`. In this way, events may be generated at any point in a dataflow network and be propagated downstream to all modules which are directly or indirectly dependent on the changed `TableView` object. Events may also be consumed by a `TableView` object if the event reflects a change to parts of the table that have been filtered out. In addition to `TableView` implementations, any class in the system may implement a `TableViewEventListener` interface to allow instances of the class to be registered as listeners.

3 Introspection of `TableView` Properties

The behavior of specific `TableView` implementations is determined by a set of *properties* which may be specified upon construction and queried or changed at run-time. A property is distinguished only by conformance to a design pattern which specifies that reading of the property be done with a method named `get`*PropertyName* and setting of the property be done with a method named `set`*PropertyName*.

For example, sample size would be a likely property of a Sample view. An implementation would define `getSampleSize` and `setSampleSize` methods for querying and changing this property. While this may seem to be a trivial point, conformance to this pattern provides a valuable benefit: the ability to introspect `TableView` objects. Introspection functionality (provided as part of Sun's JavaBeans [13] package) allows the system to discover, query, and set the values of properties of arbitrary objects at run-time. By providing this capability, introspection allows a considerable amount of functionality to be removed from individual `TableView` objects and implemented in a generic way in the environment. Introspection of **Sluice** components is currently used to automatically generate control panels for `TableView` objects, to store the configuration of dataflow networks, and to provide collaboration support.

3.1 Generating `TableView` Controls

It is often useful to construct `TableView` objects without the overhead of GUI components to control their behavior. Classes designed this way can then be used when writing applications outside of visual environments, without paying the penalty of creating and then ignoring unneeded controls. Introspection of `TableView` properties provides a way to automatically generate graphical interfaces for these "invisible" `TableView` objects. This is accomplished by obtaining a list of their properties, getting the type of each property, and presenting the user with standard editor widgets for each property. If the standard editors are not acceptable or not available for a given class, custom editors for individual properties or for the entire `TableView` object can be written, without changes or additions to the `TableView` implementation itself.

In our current implementation, for example, an integer value representing the sample size is the only property of a Sample view. This fact, along with the type of the sample size property, can be determined by the introspection mechanism at runtime. The standard editor for integer types (currently just a text field) can then be loaded by the environment and presented to the user. For aesthetic reasons, we could at some point also create a separate class that generates custom widgets for editing a sample view. However, the fact that implementation of custom editor functionality is optional greatly reduces the overhead of implementing new kinds of `TableView` objects.

3.2 Persistence of Network Configurations

A dataflow network configuration represents a lightweight view of a visualization for a dataset. For the purposes of archiving and asynchronous collaboration, it is often useful to be able to produce a readable and editable representation the configuration of the entire network, rather than just a copy of the resulting visualization. For example, analysts may wish to package a particular visualization of a dataset for viewing by others. Analysts may also simply wish to store the present state of their exploration of a dataset as captured by the dataflow network.

Java provides a generic object serialization mechanism. Serialization is the encoding of arbitrary objects into a stream of bytes. Storing the configuration of a **Sluice**-based dataflow network can be accomplished easily by simply serializing all of the objects. While this solution is ideal from the developer's perspective, the result is essentially a static, binary representation of the session. Introspection provides us with a better solution. If we can assume that the results of a session are completely determined by the original sources of data, the pattern of connections in the network, and the properties of each `TableView` module in the system, then we can produce a textual representation of the session. The MVE can generate a representation of data sources and network structure in any way it likes, and then use introspection to get the values of properties of each `TableView` object. If all property values can be converted to and parsed from strings, then a complete representation of the configuration of each `TableView` can be produced. No additional code is needed in the `TableView` implementations, and the MVE does not need to have built-in knowledge of the structure of any specific `TableView` objects.

There are several advantages of this approach over object serialization. The representation will likely be smaller, since the serialized version would contain representations of GUI components which can be ignored in a textual representation. The textual representation can also be read and (if structured properly) understood by a human. Archiving will likely be easier, and it is even possible that the textual representation could be edited externally and reloaded.

3.3 Real-time Collaboration

A primary goal of the **Sluice** framework is to support real-time collaboration over the Internet. Distributed software architectures fall 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 [4]. Replicated architectures generally require less network bandwidth, because only events are distributed instead of graphical display information. Because Internet users may be separated by relatively long network lags on low bandwidth connections, we prefer a replicated architecture for Web-based collaborative systems. We designed the **Sluice** framework to support a replicated system that only distributes change notifications, such as adding or removing `TableView`s in a network.

Ideally, modules in a **Sluice**-based environment would not be written specifically to support collaboration. Introspection of `TableView` properties supports this goal by providing an efficient way to build **WYSIWIS** (What You See Is What I See) collaborative environments using a replicated architecture. WYSIWIS refers to an abstraction for multiuser interfaces in which all collaborators see the same shared space. Stefik et al. [9] describe the degrees to which the WYSIWIS abstraction may be enforced. For example,

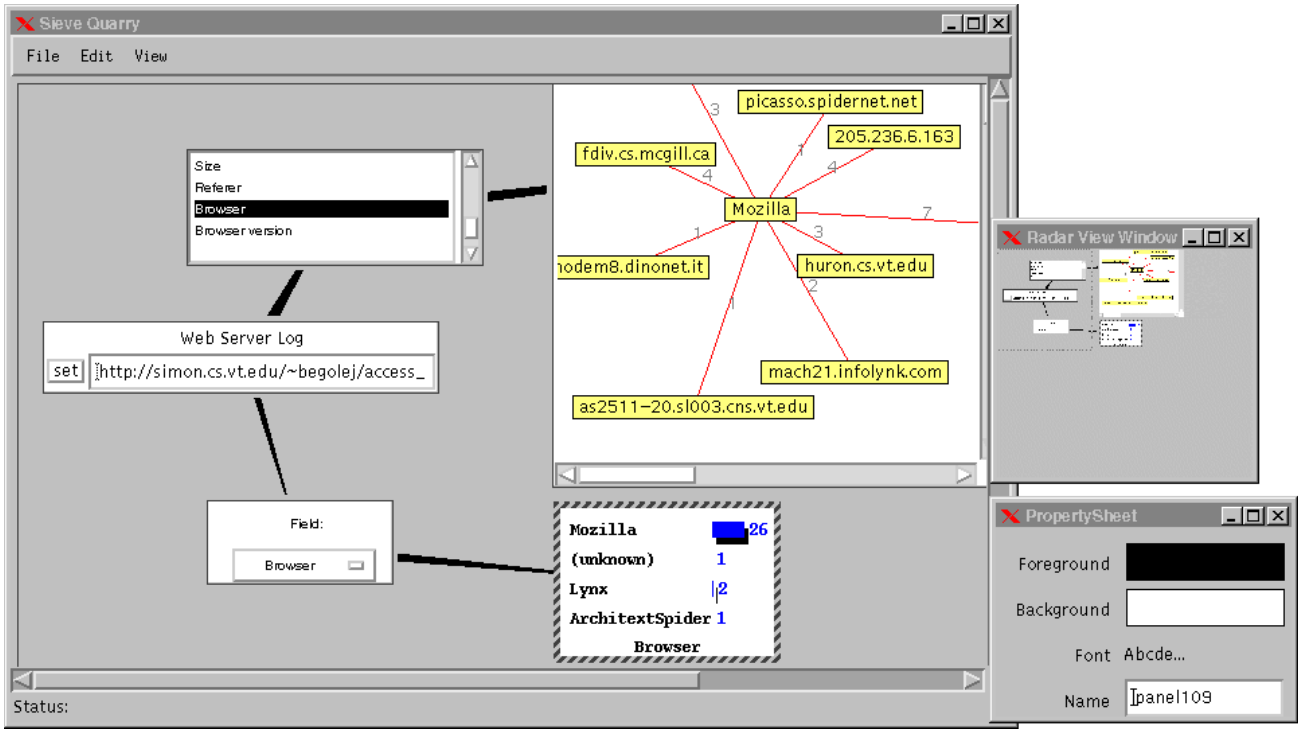


Figure 2: **Sieve** session showing two different visualizations of browser data from a Web server log.

systems may or may not allow collaborators to be looking at different parts of the workspace. From the perspective of **Sluice**, our concern is to provide means for each collaborator to see the same dataflow network representation.

A collaborative environment based on **Sluice** need only implement mechanisms for propagating dataflow network-level events like adding, linking, and removing `TableView` objects and for propagating changes to the properties of `TableView` objects and other components in the shared workspace. The latter is supported by the fact that both automatically-generated and custom property editors support the registration of listeners. This means that any object in the system may register with the editor and be notified when it changes a property. The environment can then register itself as a listener to each editor. When notified of a property change, it sends a message with a reference to the changed object, the name of the property, and the property's new value to all other collaborators. When such a message is received, the environment can find its copy of the changed object and use introspection to set the changed property to the new value. Assuming that the dataflow networks are deterministic, this mechanism will be sufficient to keep all collaborators' representations of both the structure and configuration of the dataflow network synchronized.

4 Sieve: A Collaborative Interactive Visualization Environment

We have incorporated the interactive and collaborative functionality discussed in this paper into **Sieve**, a **Sluice**-based CIMVE. **Sieve** presents the user with a large, scrollable workspace onto which data sources, `TableView` objects, and visualization components may be dropped, linked, rearranged, and edited.

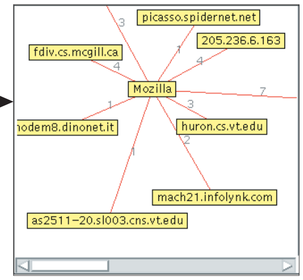
Sieve supports relaxed WYSIWIS collaboration. All collaborators have the same representation of the dataflow network, but are not forced to view the same section of the workspace at the same time. A radar pane is used to provide a "birds-eye" view of the entire workspace, and also to provide awareness of what part of the workspace other session participants are viewing.

The introspection techniques described above are used to provide editors for `TableView` objects. If a given `TableView` object is not inherently visual, its editor is placed directly on the workspace. Otherwise, the `TableView` is allowed to control its own appearance and its editor panel is displayed in a separate window when the `TableView` is selected. Placing controls and visualization output directly on the workspace consumes a considerable amount of space, but facilitates comparison of both the results of multiple visualizations and the process by which those results were obtained. For types of visualization modules whose output will not fit onto the workspace, the module can be represented by an icon in the dataflow network and the output can be presented in a separate window.

Figure 2 shows a session in the current **Sieve** prototype. The leftmost module is a source module that generates a table of data from a usage log generated by a Web server. Immediately above the source is a module which filters out all unselected fields. In this case, the user has chosen the "Hostname" and "Browser" fields. (Only the "Browser" selection can be seen in this screen shot.) The filtered data are then used by the next module in the network to generate a graph showing which hosts accessed the Web site using which browsers. Below the source is a module which counts

Field Selection

Host	*Browser*
huron.cs.vt.edu	Mozilla
fdiv.cs.mcgill.ca	Mozilla
simon.cs.vt.edu	Lynx
...	...



Web Server Log

Host	*Date*	...	*Browser*	*Document*
huron.cs.vt.edu	24/03/1997:13:19:09		Mozilla	/~cs5014/index.html
fdiv.cs.mcgill.ca	04/12/1996:21:38:43	...	Mozilla	/jasper/jasper.html
simon.cs.vt.edu	17/06/1996:02:48:24		Lynx	/JAMM
...	...		...	...

Frequency View

Browser	*Count*	*Percent of Total*
Mozilla	26	86.667
(unknown)	1	03.333
Lynx	2	06.667
ArchitextSpider	1	03.333

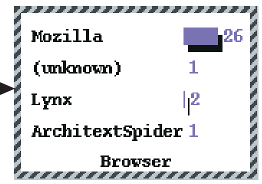


Figure 3: Table transformations from the **Sieve** session shown in Figure 2.

the occurrences of each value in a single field. The module generates a new table with a record for each value containing the value itself, the number of times that value occurred in the source table, and the percentage of the total number of records which contain the value. Figure 3 illustrates the view of the table at each of these steps.

We currently have `Table` objects in **Sieve** which support access to Web server logs, census data, and responses to the General Social Survey, a large personal interview survey of U.S. households. The two visualization applets shown in Figure 2 were adapted from sample applets distributed with Sun's Java Development Kit. We are currently converting several other freely-available visualization applets, as well as developing custom visualizations and additional `TableView` modules.

Eventually, we plan to use **Sieve** as a foundation for two learning environments being developed concurrently with **Sluice**: a collaborative virtual physics laboratory for middle and high school students, and a statistics visualization system for use by university students in the social sciences. An early prototype of the collaborative virtual physics laboratory is shown in Figure 4. This system contains primitive dataflow network creation, but requires module implementations to explicitly support collaboration. Modules must broadcast any changes made by a user to all other collaborators. In contrast, introspection of **Sluice** modules allows all collaboration support to be provided by **Sieve**, so module implementations are simpler and re-usable in non-collaborative environments.

Although its capabilities extend beyond education, **Sieve** is particularly targeted at educational applications, where real-time collaboration is particularly important. Teachers often demonstrate previous findings to students. Extending presentations to remote students gives teachers and students more flexibility, because they do not have to be physically co-located. A more exciting aspect to real-time collaborative visualization is the potential for groups of students to work through an analysis. Researchers have found that students learning together are more capable of understanding complex concepts than individually [8].

5 Conclusions and Future Work

We have presented the need for creating MVEs that support collaboration and given an overview of **Sluice**, a framework for creating these environments in Java. **Sluice**'s representation of processing as a sequence of transformations of a table of arbitrary data provides an intuitive model both for the developer of modules and for the user of environments built with **Sluice**. Simple interface specifications, design patterns, and support classes ease the task of converting existing classes for use in **Sluice**-based environments. An event propagation mechanism provides support for interaction, and the use of introspection provides an efficient means for synchronizing the dataflow networks of multiple collaborative sessions.

Our early experiences with using this framework to create **Sieve**, a prototype CIMVE, have yielded promising results. For datasets small enough to be quickly accessed over the Web, **Sluice** provides responsive interaction. A range of useful and insightful visualizations can be produced with with a small set of modules and a simple user interface. Our work on **Sieve** has also validated our assertions that the use of introspection to create control panels allows for quick creation of modules, and the simplicity of **Sluice**'s interface definitions allows for easy adaptation of existing applets and other classes for use in the system.

Given our goal of keeping **Sluice** components as simple as possible, many of our future plans are focused on exploring new ways to use **Sluice**, rather than extending the

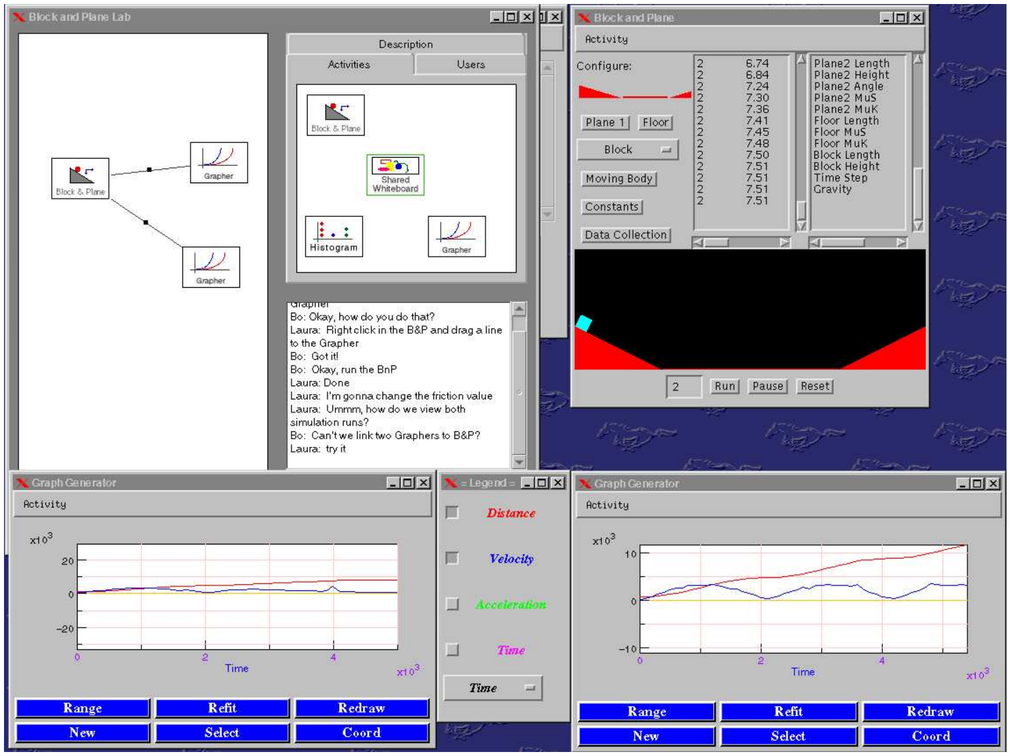


Figure 4: An early prototype of a collaborative interactive virtual physics laboratory.

framework itself. For example, we are exploring new ways of exploiting **Sluice**'s compliance with JavaBeans interface definitions and design patterns. We are also investigating extensions to the current implementation of dataflow network persistence to support scripting and archiving of network configurations. To provide the benefits of collaborative and interactive visualization without the overhead of a **Sieve**-like environment, we are exploring ways to embed **Sluice** components directly into Web pages and other containers. We are also investigating the use of `Table` and `TableView` objects as proxies for legacy software on remote systems. By this means, such legacy systems can be made available as data sources to collaborative MVEs.

6 ACKNOWLEDGMENTS

We gratefully acknowledge the support of the National Science Foundation under grant REC-9554206, and the Department of Education's FIPSE program under grant P116-B60201.

REFERENCES

[1] G. Abram and L. A. Treinish. An extended data-flow architecture for data analysis and visualization. In *Proceedings of Visualization '95*, pages 263–270. IEEE, Oct. 1995.

[2] N. Alper and C. Stein. Geospatial metadata querying and visualization on the www using Java (tm) applets. In *Proceedings of Visualization '96*, pages 77–84. IEEE, Oct. 1996.

[3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, 1996.

[4] J. Begole, C. A. Struble, and C. A. Shaffer. Leveraging Java Applets: Toward Collaboration Transparency in Java. *IEEE Internet Computing*, March-April 1997.

[5] H. H. Clark and S. E. Brennan. Grounding in communication. In L. B. Resnick, J. Levine, and S. D. Bahrend, editors, *Perspectives on socially shared cognition*. APA, 1990.

[6] D. S. Dyer. A dataflow toolkit for visualization. *IEEE Computer Graphics and Applications*, 10(4):60–69, July 1990.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[8] T. Koschmann, editor. *CSCL: Theory and Practice*. Lawrence Erlbaum Associates, Mahwah, NJ, 1996.

[9] 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. Apr. 1987.

[10] Java (tm) Remote Method Invocation Specification. URL: http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi.pdf, November 1996.

[11] JDBC (tm): A Java SQL API. URL: ftp://splash.javasoft.com/pub/jdbc-spec-0120.pdf, January 1997.

[12] Java (tm) IDL. URL: http://splash.javasoft.com/JavaIDL/pages/index.html, 1996.

[13] JavaBeans (tm). URL: http://splash.javasoft.com/beans/beans.100A.pdf, December 1996.

[14] C. Upson, T. A. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The Application Visualization System: a computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.

[15] Visual Numerics JWAVE. URL: http://www.vni.com.

[16] G. J. Wills. Selection: 524,288 ways to say "this is interesting". In *Proceedings of Visualization '96*, pages 54–60. IEEE, Oct. 1996.

[17] J. D. Wood, K. W. Brodlie, and H. Wright. Visualization over the world wide web and its application to environmental data. In *Proceedings of Visualization '96*, pages 81–86. IEEE, Oct. 1996.