Jadve: A Framework for Graph-Based Data Visualization Applications

Wenke Lee
Computer Science Department
Columbia University
New York, New York 10027
wenke@cs.columbia.edu

Naser S. Barghouti
Bear Stearns & Co.,
575 Lexington Ave.,
New York, NY 10022
naser@bear.com

Abstract

This paper presents the design and implementation, as well as example applications of *Jadve*, a framework for graph-based data visualization applications written in Java. The main contributions of this work are: compared with traditional graph editor frameworks, *Jadve* has built-in distributed processing and multi-user collaboration support; by utilizing Java's dynamic and portable features, applications can be easily constructed on top of *Jadve* and be able to run on virtually all platforms. *Jadve* provides a Java application programming interface (API) for programmers to build client-server applications that need to visualize information in terms of graphs, such as business workflows or program dependencies. Through subclassing, the classes that implement a specific application inherit, customize, and enhance the *Jadve* graph representation, presentation and layout services. *Jadve* employs a client-server architecture: the front-end client supports interactive editing, display, and browsing of graphs; whereas the back-end server provides graph layout, storage, concurrency control and collaboration management services. Three applications have been built on top of *Jadve*: a new version of *Improvise*, a process modeling environment; *Chava*, a graphical navigator for Java programs and document repositories; and *Classifier Visualizer*, a data classifier display and analysis module within a data mining environment.

1 Introduction

Data visualization is a very important and viable tool to show the structure of technical information. Various data visualization technologies have been developed for different application domains. For example, computer mapping is now an integral part of Geographical Information Systems (GISs); numerical data display (e.g., curves, bar charts, and pie charts, etc.) is widely used in business and scientific computing; directed attributed graphs (DAGs) representing the semantics of the underlying data are also used extensively in applications such as process (workflow) modeling, program visualization and understanding, and data mining.

An interesting example of graph-based data visualization applications is *Improvise* [BKC95], a multimedia system for modeling, visualizing and documenting software and business processes and workflows. It represents a process flow as a multi-layered directed graph and provides a graphical user interface (GUI) for displaying, editing and browsing the graphs. *Improvise* uses nodes and edges that have different shapes and styles to represent different types of process entities. For example, a box (rectangle) with two horizontal lines in it represents a **manual task**, and an ellipse represents a **computer process**. It provides an object selection template that contains the different types (shapes and styles) of nodes and edges so that the user can insert a node or an edge to the graph by simply selecting an object from the template and clicking at a point (location) on the canvas. The user can also annotate each node and edge with multimedia information, e.g., video clips, and executable attachments by defining its attributes. In many simple graph drawing environments, the user has to make a lot of effort to consider the size and placement of each node and edge in order to make a good presentation of the graph. *Improvise* completely eliminates such burdens by using *dot* [KN93], a graph layout generator, to compute the graph drawings, i.e., the size and placement of each node and edge. Thus the user only needs to place the nodes and edges on the canvas casually, and can issue a layout request to obtain high quality graph drawings.

From *Improvise* and other advanced systems, we see that a graph-based data visualization application typically includes these basic modules (functions):

- Graph Representation. It allows the user to define graphs that represent application-specific information.
- Graph Presentation. It displays the graphs using a graphical user interface (GUI), and allows the user to edit and browse the graphs.
- Graph Layout. It generates graph drawings automatically using algorithms that consider graph characteristics (e.g., planarity, orientation, etc.) and aesthetic criteria (e.g., maximization of the display of symmetries).

With the increasing importance of Web-based, and more generally, distributed computing, many applications also need to have these more advanced features: remote graph drawing services and multi-user collaboration support. By remote graph drawing, we mean the ability for an application running on a client machine to request graph drawing (layout) services from a server running on another machine anywhere on the network. By collaboration, we refer to the situation where multiple users from different machines on the network share the same set of graphs for editing and viewing.

This paper presents *Jadve*, a framework for graph-based data visualization applications. The motivation of building *Jadve* is to not only provide the standard (built-in) graph representation, presentation, (remote) layout services and collaboration support, but also facilitate rapid application development. *Jadve* thus provides an API so that its standard services can be customized to accomplish the application-specific tasks.

Jadve employs a client-server architecture. The *Jadve* server provides collaboration support and employs *dot* to compute graph layout; the front-end client supports interactive graph editing and browsing, and can be launched from any node in the network and be connected to the server. This architecture has the advantage of saving computing resources since the computational-intensive graph layout process need not be on every client machine; in addition, graph representation can be stored by the server and shared by multiple clients.

The advent of Java [AG96] motivated us to exploit the features it provides, such as portability, graphical user interface generation facilities, and distributed computing features. We implemented *Jadve* in Java, which allows the frond-ends of *Jadve*-based applications be built as applets that can be executed over the WWW via any Java-enabled Web browser; this greatly facilitates the distribution of the application software and enhances its availability.

The rest of this paper is organized as follows: Section 2 discusses the design and implementation of *Jadve*. Section 3 describes the *Jadve* API. Section 4 presents three example applications developed on top of *Jadve*. Section 5 compares *Jadve* with other related frameworks. Section 6 concludes the paper with an evaluation and a discussion of future work.

2 The Design and Implementation of *Jadve*

In this section, we first discuss the overall design goals of *Jadve* and the advantages of using Java technologies for its implementation. We then discuss the design considerations of its main functionalities, namely, graph representation, presentation, layout and collaboration support. We describe the implementation of *Jadve* by presenting its main components.

2.1 Design Goals

From a framework technology point of view, we had three main requirements in building *Jadve*: extensibility, portability, and customizability. Extensibility allows for a natural evolution path, where new services can be incorporated into the framework. Portability is essential in the multi-platform world we live in. Customizability allows for many different applications to be built on top of the framework, re-using its services and avoiding re-writing a lot of code.

In the following sections we consider how to accomplish these design goals and the required functionalities (services). Choosing the right language for implementation is the important first step because the strengths and weaknesses of the language affect the detail design of the system. We used Java because of these features:

- Object-oriented. Adopting an object-oriented programming paradigm often results in a good component-based software architecture, which is essential for the extensibility of a system. Moreover, object-oriented design is natural for *Jadve*, which is graph-based. A graph has a composite structure of graph objects, i.e., nodes, edges, and sub-graphs. More importantly, there is a natural inheritance hierarchy among graph objects of different shapes and styles. For example, an edge with an arrow head can be derived from a plain edge (a straight line with no arrow).
- Distributed. The Java Development Kit (JDK) includes networking library to use sockets and TCP/IP protocols such as HTTP and FTP. These Java functions are essential for the multi-user collaboration support services.
- Architecture Neutral and Portable. These Java features ensure that *Jadve* can run on any computing platform that supports the Java Virtual Machine. In other words, *Jadve* is portable.
- Multi-threaded. It enables *Jadve* to perform more than one task for the user at the same time period. For example, when the user requests a graph layout, a separate new thread (in the background) contacts the server and waits for the reply. At the meantime the original thread continues to interact with the user.
- Dynamic. In Java, finding out run time type (class) information is straightforward. Therefore new classes can be easily added to a running program by supplying their full class names, which indicate the search paths for the classes, to the program. Using this Java feature, the *Jadve* API essentially facilitates the integration of application-specific functions (classes) into *Jadve* in runtime, thus enables *Jadve* be customized for specific applications.

2.2 Graph Representation, Presentation and Layout

Graph representation deals with using graph to capture the semantics of application data. The nodes and edges of the graph are therefore application data entities. Applications normally use different symbols to represent different types of application data. In *Jadve*, these symbols are just template nodes and edges, each of which has a different shape and drawing style.

For graph presentation, the *Jadve* front-end displays the graphs and provides functions to edit, i.e., add, delete, copy, and paste, a node or edge; browse, i.e., zoom and scroll, the graph; and analyze the graph, e.g., find all reachable nodes from a given node. Some of these functions, e.g., the browse functions, are generic in the sense that they have the same (well-defined) behavior and consequences across all applications. Whereas other functions can have additional application-specific semantics. For example, an application may have different data-driven processing when a **computer process** is deleted than when a **manual task** is deleted. Therefore these functions need to include two parts: the standard operations, and the application- and data type- specific operations.

For graph layout, the *Jadve* front-end simply provides a command for the user to request a layout. The layout procedure is generic to all graphs since it computes the drawing attributes, i.e., the sizes and placements, of the nodes and edges by considering only their shapes, labels and connection information.

We can see from the above discussion that graph objects, namely, graphs, nodes, edges, and their attributes, are the central elements in *Jadve* because its basic services are essentially operations on these objects. There are both general and application-specific graph representations and operations in *Jadve*. We can conceptually think of two types of graph objects: abstract objects and application objects. Abstract graph objects are the defining elements of a generic graph, i.e., the subgraphs, the nodes, and the edges. Whereas the application objects are the results of overlaying application semantics on top of the abstract objects. It is important to realize that it is neither possible nor appropriate to design *Jadve* to include, as built-in, all possible application-specific graph objects (their representations and operations) since it is intended as an application framework on top of which a wide varieties of systems can be developed. The challenges are to design *Jadve* in a such a way that application objects and operations can be seamlessly integrated with its built-in generic graph representation, presentation and layout services.

2.3 Multi-user Collaboration Support

Collaboration in data visualization is normally in the form of sharing (reviewing) the edit of a graph by others. There are many possible application-specific scenarios, for example,

- Before a user can make changes to a graph, he/she is required to first obtain an exclusive lock for it so that no other users can make changes to the same graph at the same time. The lock is released once the user deposits the changed graph to the master repository.
- Before a user, Joe, makes changes to a graph, he first checks out (downloads) the latest version of the graph from the master repository. No lock is placed on the graph. When Joe deposits the changed graph, if another user, Diana, has already deposited her own changes (to the same graph), Joe and Diana needs to negotiate and merge their changes.
- Before a user can deposit his/her changes, he/she needs to present the changes to all other users and get their approvals.

It is thus evident that *Jadve* should not be hard-coded to support a fixed collaborative workflow. Rather, it should provide these general primitives and support mechanisms:

- File management: provides a repository for (shared) graph files; supports the lock/unlock and check-out/deposit operations.
- User management: defines a group as a set of users that collaboratively work on a collection of graph files (it is assumed that no files are shared between two groups); supports communications within a group, e.g., broadcasts a message to all users, or sends a message from one user to another.

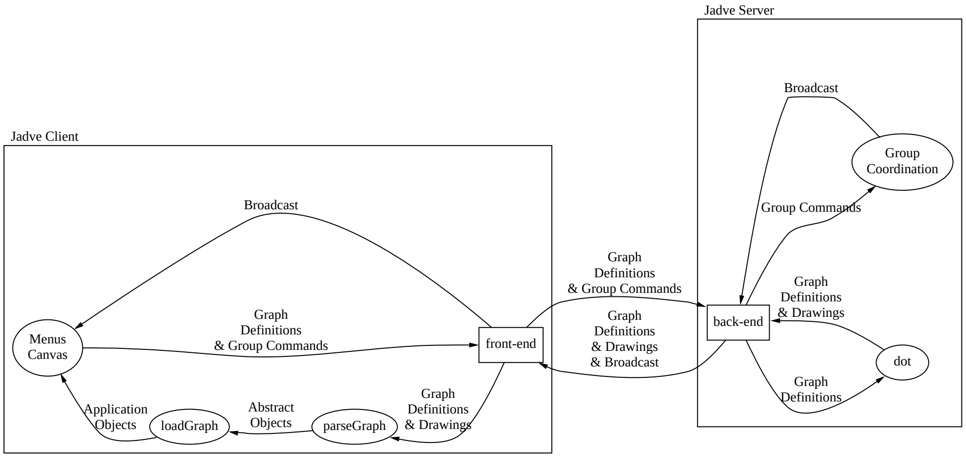

Figure 1: The Components of *Jadve*

- Application-specific concurrency control: enforces a group-specific policy, which is a sequence of operations that must all succeed before a deposit can be made to the repository. For example, a policy with a single operation, "no", indicates that no concurrency controls is in place (that is, the Unix semantics is assumed); whereas, a policy, "lock;review", specifies that a lock must be obtained first, and a group review/approval process needs to take place.
- Utilities: a function that compares two graphs and highlights the differences; a "review" viewing facility that allows a user to display, in real-time, his/her changed graph remotely to other users in the group.

2.4 The Components of *Jadve*

In this section, we present the architectural components of *Jadve*, as shown in Figure 1 (generated using *dot*). The ellipses in the figure represent the modules (classes or methods), and the directed edges with their labels represent the data flowing between the modules. The two boxes, "front-end" and "back-end"', represent the main (container) modules of the *Jadve* client and server respectively. The server and the client are two different programs that communicate through the networks.

2.5 Graph Object Definitions

We first discuss the implementation of graph objects, which is the foundation of all *Jadve* components.

Figure 2 shows the definitions of *Jadve* abstract objects. The solid edges in the figure represent the composite relations. The dotted edges represent the inheritance relations, e.g., *DotNode* is extended from *DotElement*. Here *DotGraph* defines a graph as having a set of attributes, nodes, edges, and subgraphs. A graph element, a node or an edge, is defined by *DotElement*, which also has a set of attributes. *DotAttribute* defines an attribute as a $(name, value)$ pair. Some details are not shown in the figure. For example, each *DotElement* has a pointer to its container, *DotGraph*; and each *DotEdge* has pointers to its two end-nodes. Using these pointers, the composite structure of the entire graph can be traversed starting from any of its nodes or edges.

The abstract objects only describe the compositions of a graph. The application objects are the result of adding application representation and operation semantics to the abstract objects. As an application framework, *Jadve* facilitates the definitions of application objects using the class hierarchy shown in figure 3. Again, the solid edges in the figure represent the composite relations and

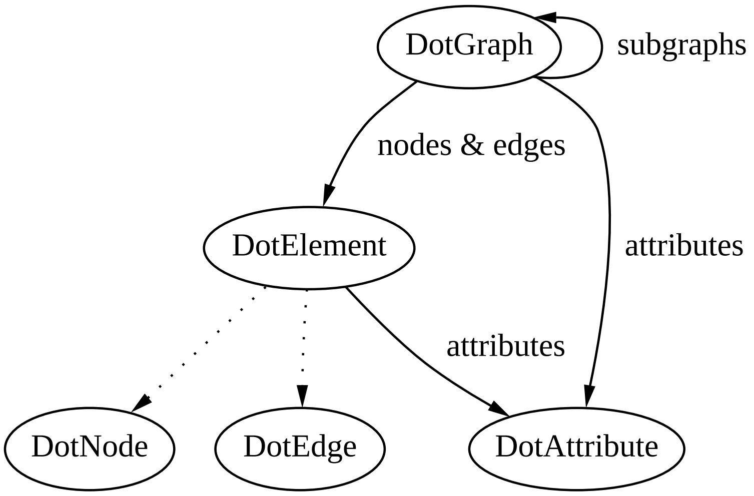

Figure 2: *Jadve* Abstract Objects

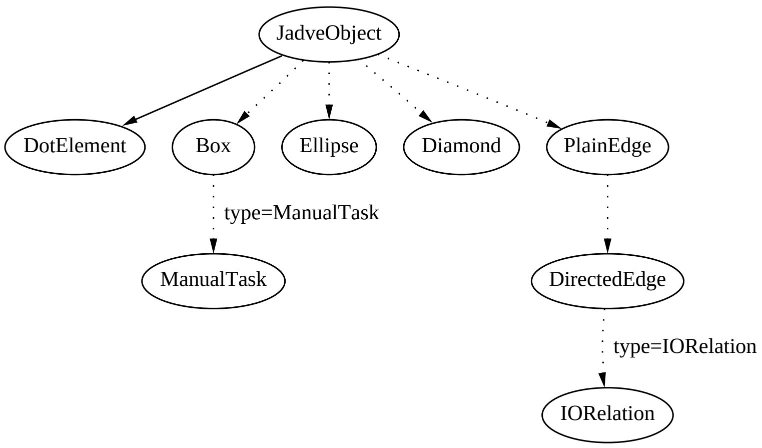

Figure 3: Class Hierarchy of *Jadve* Application Objects

the dotted edges represent the inheritance relations. *JadveObject* defines the most general aspects of an application object. The one-to-one correspondence between an application object and its abstract object is represented by the reference to its *DotElement*. *JadveObject* defines methods for drawing, locating (deciding whether the mouse position is within the object boundary), highlighting (painting the object in red), scaling (enlarging/shrinking the object when the view is zoomed in/out), and translating (modifying the object position when the view is scrolled), etc. The *draw* method is overridden by the subclasses, e.g., *Box*, which have detailed information about the shapes of the objects. *Jadve* also includes, as built-in, a number of subclasses of *JadveObject*, each with a commonly used shape and style. These include *Box*, *Ellipse*, *Diamond*, *PlainEdge*, and *DirectedEdge* (an edge with an arrow head), etc. Application-specific graph objects are derived from *JadveObject* or one of its subclasses. For example, in *Improvise*, *ManualTask* is implemented by subclassing *Box*. It extends (overrides) the *draw* method of *Box* by adding two horizontal lines to divide the rectangle into three sections, and displaying the task *name*, *description*, and *organization* in the three sections respectively.

Here we have separate definitions for abstract objects and application objects. There are alternative approaches for implementing the graph objects. For example, *JadveObject* and *DotElement* can be combined into a single class hierarchy (the combined class is still part of the *DotGraph* composition hierarchy). The motivation of our approach comes from the principle of separation-of-concern. *DotGraph* and *DotElement* deal with defining a graph; whereas *JadveObject* and its subclasses deal with the representation of application semantics. Customizing *Jadve* normally only involves extending the *JadveObject* classes.

2.6 The *Jadve* Client

The *Jadve* client handles most of the interactions with the end-user. Its graphical user interface is constructed using the standard Java AWT (Abstract Windowing Toolkit), with a template of application objects, pull-down menus for edit, display and analysis commands, and canvasses for displaying (drawing) graph objects.

As the front-end to application users, the *Jadve* client provides the seamless integration of application-specific graph representations and operations with the generic *Jadve* services. Using *Improvise* as an example, when the user adds a **Manual Task** to a process flow graph, a *ManualTask* object is created and added to the list of graph objects (*JadveObjects*) displayed on the canvas (defined as *JadveCanvas*). In addition, its corresponding *DotElement* is created, with attributes $(shape, box)$ and $(type, ManualTask)$, and added as one of the *nodes* of the *DotGraph* associated with the *JadveCanvas*. When the user requests a layout, the *writeGraph* method of *DotGraph* writes out the graph definitions (in text form) according to the *dot* syntax. A *layout* request along with the graph definitions are then sent to the server. When the client gets back the graph drawings (also in text form and in the *dot* syntax), the *Jadve* parser, *parseGraph*, reads the drawing descriptions, and constructs a new *DotGraph* accordingly. Each new *DotElement* is just the old one (before layout) with the new *size* and *position* attributes. To transform these new abstract objects to the corresponding application graph objects, the *loadGraph* method of *JadveCanvas* utilizes the dynamic feature of Java to first look for the *type* attribute of each *DotElement* object, and then create a new object instance of *type* (if the *type* attribute is null, as in the vanilla *Jadve* environment, the *shape* attribute is used instead). For example, if a *DotElement* is of type "ManualTask", then its corresponding new *JadveObject* is a *ManualTask* object. The last few steps of *loadGraph* are to erase the list of old graph objects, and display the new objects on *JadveCanvas* by invoking the *draw* method of each *JadveObject*. From this point on, the graph objects on the canvas carry the application representation and operation semantics. Section 3 describes how application-specific object operations are supported.

Jadve client also provides support for multi-user collaboration. Users in a group coordinate their operations by passing proper messages. Each client has a thread that sends/receives messages to/from other clients (by using the server as the relay, see Section 2.7). For example, when the concurrency control policy requires the "review" operation, a user, Joe, can let others review his edited graph by broadcasting a "review" message to the group. Upon receiving this message, each client creates a review view displaying Joe's changed graph for the corresponding group member. These review views are the slaves of Joe's workspace view, where he has made his changes. They have exactly the same display as Joe's view even when it zooms, scrolls, or highlights an graph object. Joe can therefore "walk" the reviewers through his specific changes.

The *Jadve* client has a tool called "diff". It compares the graph on the canvas with another graph (not displayed) and highlights the deleted, inserted and modified graph objects using different colors. This command can be used to show the differences between two versions of the same graph. For example, after failing to deposit his changes because user Diana has just updated the same graph, user Joe can use the "diff" command to compare his version with Diana's version. He can then try to merge all the changes and deposit again.

2.7 The *Jadve* Server

The main function of the *Jadve* back-end concurrent server is to process client requests for graph layout and group communications. It listens to a socket for incoming messages and creates a Java thread to process each message. The header of each message between the server and the client specifies its message type. For a *layout* message, the server invokes the *dot* process to compute a graph layout. As shown in figure 1, the server passes to *dot* the message body, the graph definitions, and sends the *dot* output, the graph definitions enriched with drawing information, back to the client.

The server supports multi-user collaboration according to the configuration file that defines the groups, and for each group, the users, graph files, and concurrency control policy. When a client sends a *register* message, the server first checks whether the user belongs to the group he/she intends to join. It then records the client's IP address and port number. Knowing the members of the group and the IP addresses and port numbers, the server can use sockets to relate group message from one member to the others in the group. For example, during the review session, when user Joe, the member who has requested the review, zooms in the graph, the corresponding client sends a *group command* message to the server. The server then sends a *broadcast* message with the same message body, which is the command "ZoomIn", to other clients. Upon receiving such a broadcasted command, each client executes it on its review view so that the group member sees exactly the same graph display as Joe.

The *Jadve* server server also manages the master (centralized) repository of graph files, and supports the lock/unlock and check-out/deposit operations. The lock/unlock operations are straightforward since the server is the only process that can access the files in the master repository. The server maintains a list where each entry contains a file and the user that holds the lock. For each check-out or (successful) deposit, the server also records the user ID and the timestamp. When user Joe attempts to deposit a graph, if the server discovers that Joe's check-out timestamp of the graph is smaller (earlier) than user Diana's deposit timestamp, it rejects Joe's deposit. The server sends a "reject" message, along with the "new" graph (updated by Diana) to Joe. The corresponding client then automatically invokes the "diff" command to show Joe the differences between the two versions. The server also deletes Joe's old check-out entry and creates a new one with the current timestamp, so that Joe can deposit after merging the two versions.

3 The *Jadve* API

The purpose of the *Jadve* API is to provide a means for developers to extend and customize *Jadve*. Using this API, an application-specific *Jadve* class is defined as the subclass of a built-in *Jadve* class. Application-specific behavior can then be implemented by adding new data member and methods to the built-in class, and overriding its existing methods. This section only gives high level descriptions of the essential *Jadve* classes, and is not intended to be a complete reference manual.

As described in Section 2.5, application-specific graph objects can be defined by subclassing *JadveObject* and implementing their own *draw* functions. Most applications require functions to analyze the structure of a graph. From an application object, *JadveObject*, its reference to the corresponding abstract object, *DotElement*, is an entry into the graph composition hierarchy. *DotGraph* provides traversing methods to visit every element of a graph.

JadveCanvas implements the canvas. Application objects displayed on the canvas can be accessed through two methods: *getListOfObjects* returns the list of all application objects; and *getSelectedObject* returns the selected (highlighted) object. To modify the behavior of *Jadve* built-in graph operations (e.g., *delete*, *zoom*, etc.), application-specific actions can be defined in the *Jadve_eventHandle* method, which is invoked whenever a *Jadve* built-in command is issued. Its default implementation is an empty function, thus allowing the standard operations to take place.

JadveMenu implements the pull-down menus. Menu commands that operate on graph objects are passed to the *JadveCanvas* attached to the menu bar. The *addMenu* and *addMenuItem* methods allow application-specific commands, i.e., commands not provided as built-in by *Jadve*, be added to the pull-down menus. For each application command, e.g., "Import", a corresponding class,

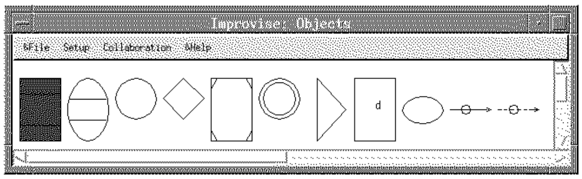

Figure 4: *Improvise* Template Objects

e.g., *Import*, needs to be implemented as a subclass of *JadveAppCommand*. The actual command actions are specified in its *execute* method, which is invoked when the command is selected from the pull-down menus.

JadveObjectTemplate implements the object template window. Similar to *JadveCanvas*, it has the *getListOfTemplates* and *getSelectedTemplate* methods for accessing the template objects, each of which is a *JadveObject*. *addTemplate* and *deleteTemplate* can be used to add and delete a template object respectively.

JadveClient implements the front-end client. It contains a *JadveObjectTemplate*, and a number of *JadveMenus*, each of which has an attached *JadveCanvas*.

JadveServer implements the back-end server. Its *addMessageType* method enables the server to process messages other than *layout* requests and group commands. Similar to *JadveMenu*, for each new message type, e.g, "executeChava", a corresponding class, e.g., *executeChava*, needs to be implemented as a subclass of *JadveAppMessage*. The actual message handling actions are specified in its *execute* method, which is invoked whenever the server gets such a message.

Finally, in a single-user environment, the only service that the *Jadve* server provides is the graph layout, therefore it is desirable that the *Jadve* client has the option to invoke the *dot* process locally (thus eliminating the need to communicate with the server across the network). *DotLayout* is provided to support single-user applications. It has methods for parsing graph definitions and invoking *dot* to layout a graph. The *doLayout* method of *JadveCanvas*, which by default sends a *layout* message to the server, can be overridden to instead use the *DotLayout* methods.

4 *Jadve* Applications

In this section, we describe briefly several examples to demonstrate how various data visualization applications can be constructed on top of *Jadve*. Note that the vanilla *Jadve* environment is different from these applications in that it contains no application-specific graph objects and operations. For example, its object template window contains only the built-in *JadveObjects*, i.e., *Box*, *Diamond*, etc.

4.1 Example 1: *Improvise*

The Java version of *Improvise* is the first comprehensive customization of *Jadve*. In fact, *Jadve* was extended to include new components (e.g., the object template window) once we realized that some *Improvise* functions need general support from the framework.

Improvise is a process (workflows) modeling and analysis environment. Each type of *Improvise* workflow objects has its own special shape and drawing style. Therefore a rich set of application objects, i.e., *ManualTask*, *ComputerProcess*, *IORelation*, etc., are implemented. *ImproviseObject-Template* is extended from *JadveObjectTemplate*. It replaces the built-in *JadveObjects* with the entire set of *Improvise* application objects, see figure 4.

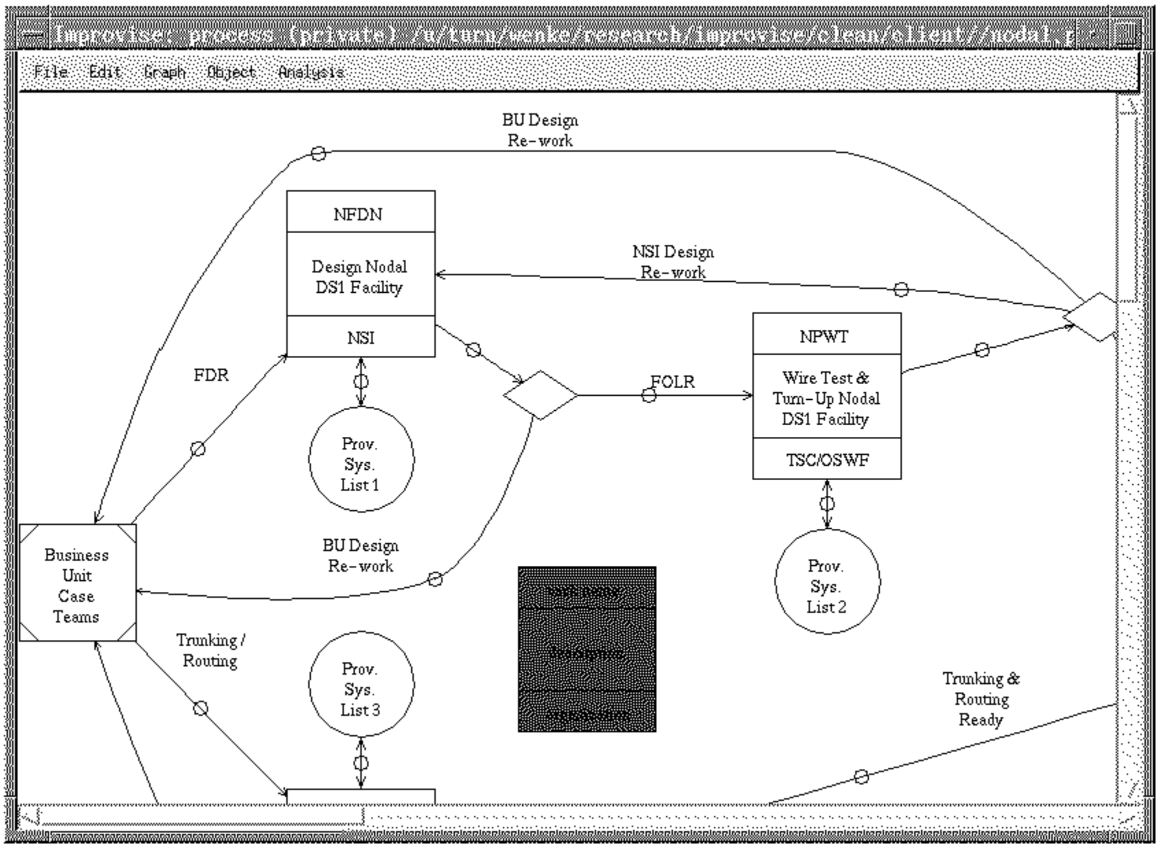

Figure 5: *Improvise* Menu and Canvas Showing a Workflow

Figure 5 shows the *Improvise* menu and canvas displaying a workflow graph. The user has just added a *ManualTask* to the graph, by selecting it in the object template window and clicking an unoccupied position on the canvas. This new object is automatically highlighted (selected) so that any subsequent object command is operated on it. For example, the "attribute" command (in the "object" pull-down menu) can be used to specify its attributes, e.g., *name*, *organization*, etc.

Improvise has a set of process analysis commands. For example, when the "expand graph" command is issued upon a node that represents a subgraph (a next layer process graph), the entire subgraph takes the place of the node in the current graph. This command requires complex graph traversing operations, and is not implemented in *Jadve*. To add this command, *ImproviseMenu* is implemented as a subclass of *JadveMenu*. It has the "expand graph" menu item added to the "graph" pull-down menu. An instance of *expandgraph* (a subclass of *JadveAppCommand*) is created when the *ImproviseMenu* is instantiated. The constructor of *expandgraph* has, as input parameter, a reference to the *ImproviseMenu*. From this reference, *expandgraph* methods, e.g., *execute* and *expandInPlace*, can access the *JadveCanvas*, and hence the *Improvise* objects and workflow graphs.

Improvise is multimedia-enabled. That is, some of the *Improvise* objects can have associated multimedia (audio and/or video) data. Two new commands, *playAudio* and *playVideo*, are added to the "object" pull-down menu. These commands are disabled unless the highlighted object has the appropriate media files. The *Jadve_eventHandle* method of *ImproviseCanvas* (a subclass of *JadveCanvas*) specifies these *Improvise*-specific operations for the built-in "hightlight" event (triggered when a graph object is selected): if the "audio" attribute of the highlighted object is not null, enable the "play audit" command; if the "video" attribute is not null, enable the "play video" command. *playAudio* and *playVideo* are implemented each as a subclass of *JadveAppCommand*. Their *execute* methods implement the operations to play the media of the highlighted object.

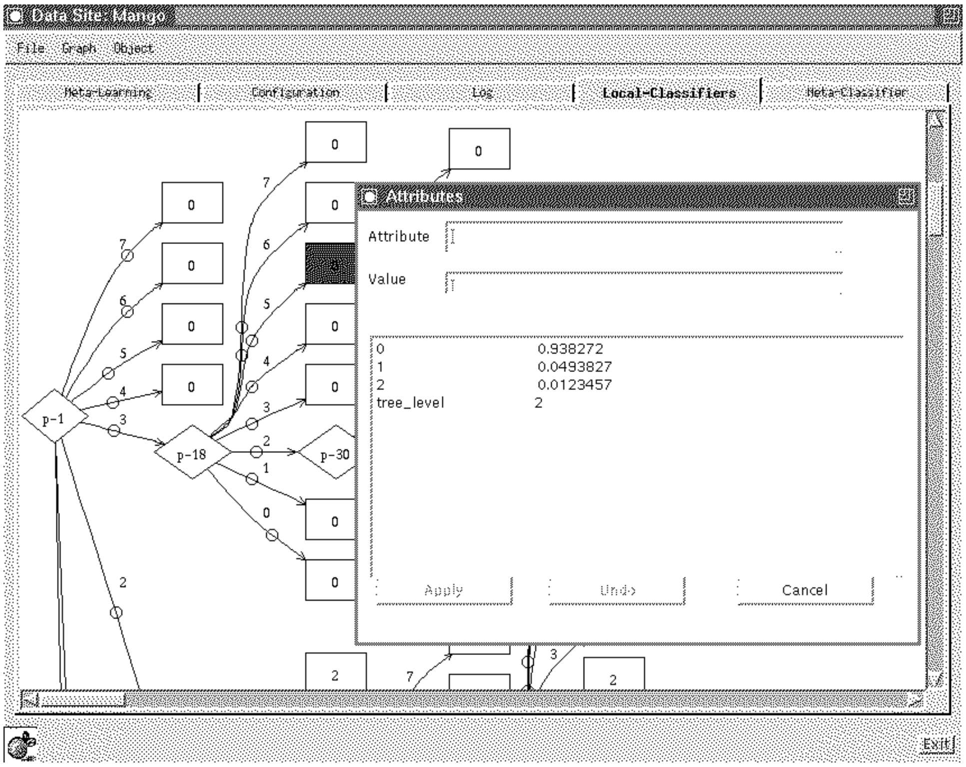

Figure 6: The Classifier Visualizer

Improvise also fully utilizes the built-in multi-user collaboration services of *Jadve* to support share editing and viewing of process graphs.

4.2 Example 2: *Chava*

Chava [KK96] is a graphical navigator for analyzing the structures (e.g., class hierarchy, or message-passing relations among classes) of Java programs. It was originally implemented using *lefty* [KD91] and *ciao* [CFKW95].

To use *Chava*, the user needs to first run a set of *Chava* scripts in the directory containing the Java programs. These scripts analyze the programs and build databases of Java program entities. A graphical front-end can then display the program entities and their relations.

The *Jadve* version of *Chava* was a simple customization. A new command *runChava* (a subclass of *JadveAppCommand*) is added to the *JadveMenu*, and a new message type of *executeChava* (a subclass of *JadveAppMessage*) is added to the *JadveServer*. The *runChava* object on the client side is responsible for sending an *executeChava* message, which contains the directory path of the Java programs to be analyzed, to the server. The *executeChava* object on the server side runs the *Chava* scripts, and sends the output (which is already in *dot* format) back to the client for display.

Compared with its original implementation, the benefit of using *Jadve* is that *Chava* users can take advantages of the group collaboration features, for example, to review a Java program together.

4.3 Example 3: Classifier Visualizer

An important issue in Knowledge Discovery and Data Mining (KDD) is the presentation of the discovered knowledge [FPSS96]. A human expert needs to understand and inspect (judge) the knowledge before it can be made useful (e.g., put into a report). A large number of data mining algorithms produce classifiers, which can classify a data item into one of the pre-defined categories (labels). There are many kinds of data classifiers, e.g., a decision tree, that can be represented as a DAG. We therefore wanted to apply *Jadve* to this application domain.

Figure 6 shows a decision tree, where the leaf nodes represent classes (decisions), the non-leaf nodes represent the attributes under test, and the edges represent the attribute values. The "Attributes" window in this figure shows the following decision information about the highlighted leaf node (with class label "0"): for a data item, if its "p-1" value is 3 and its "p-18" value is 5, then it belongs to class "0" (with .938 probabilities).

The Classifier Visualizer is an example of utilizing only some of the components of *Jadve*. Here, the *JadveCanvas* is "plugged" into a host data mining environment, *JAM* [SPT+97]: instead of using *JadveMenu*, the graph menu items were added to the menu bar of *JAM*. When a graph command, e.g., "Zoom In", is selected, the corresponding method, e.g., *zoomIn*, of *JadveCanvas* is invoked. The *doLayout* method was overridden to use methods in *DotLayout*, instead of communicating with a *JadveServer*. A "load" command is also added to transform data classifiers into graph descriptions, and invoke *doLayout* to obtain the graph drawings.

5 Related Work

Related extensible graph editors include *EDGE* [New88], *dotty* [KN94], and *HotDraw* [Joh93]. These systems all treat pictures of graphs as structured objects, support a general set of menu-driven graph viewing and editing operations, and allow programmers to add application-specific functions. *EDGE* and *dotty* also provide on-demand graph layout services.

dotty has two main components: *lefty* as the front-end and *dot* as the back-end. The *lefty* process runs a program written in the *lefty* (procedural) scripting language. This program includes functions to edit graph objects and to draw the objects according to their attributes (e.g., color, shape, and style). It also has a graph layout function that communicates with the *dot* process running in the background. Customizing *dotty* involves directly (and freely) changing the *lefty* program (since the source code is always accessible to the user). For example, the *leftdown* function is called when a user presses the left mouse button. It by default calls *insertnode* to insert a new node in the position of the cursor. The user can rewrite the *leftdown* function to perform other operations, for example, to pop up a menu. It can be argued that *dotty* gives complete freedom to programmers to build applications since they can access and modify all functions. However customizing *dotty* may have large overhead because one needs to understand the internals of the *lefty* program, and to learn *lefty*, a non-standard language. *Jadve* provides all of the functions supported by *dotty*. In addition, by providing a standard *Java* API, it hides the implementation details from the application programmers, and guarantees a degree of uniformity in terms of user-system interaction across different applications.

HotDraw is a framework for structured semantic drawing editors. It is written in Smalltalk, and is therefore object-oriented. The most important class in *HotDraw* is *Figure*, which is the super class of all drawing elements in a *HotDraw* application. Each subclass of *Figure* redefines its drawing methods and attributes to represent application-specific information. Application programmers can further customize *HotDraw* by creating subclasses of (and making changes to) the important classes such as *Drawing* (which represents the entire drawing), *Tool*, *Handle* and *DrawingEditor*. A major difference from *Jadve* is that *HotDraw* has built-in support for semantic operations. The elements of a semantic drawing can have constraints on their behavior, for example, an attribute of one element is a function of the attributes of other elements. Constraints in *HotDraw* are implemented as objects.

A *Constraint* object is created if a constraint is needed between two figures. It gets notification from the constraining figure and changes the constrained figure to satisfy the constraint. *Jadve* can in principle be extended to support constraints. The constraining *JadveObject* can have a *DotAttribute* (“*constraint*″, *constraint_name*) specifying the name of (a subclass of) *JadveConstraint*, and a list of (“*constraining*″, *element_id*) specifying the list of constrained graph objects. As the triggering operation is performed on the constraining object, the *Jadve_eventHandle* method of *JadveCanvas* can invoke the *execute* method of the corresponding *JadveConstraint* object, which in turn traverses all constrained objects and makes proper changes.

It should be noted that *Jadve* is not just an *editor*. Its client/server architecture, multi-user collaboration support, as well as the use of Java, are all aimed to provide a framework for building more advanced distributed applications.

6 Conclusion

The two basic requirements of an application framework are: first, it needs to provide a set of standard services valuable to the application domain; second, these services can be extended and customized to accomplish application-specific tasks.

In this paper we presented *Jadve* as a framework for graph-based data visualization applications. *Jadve* implemented, as built-in, a set of graph representation, presentation, and layout services, as well as multi-user collaboration support. To facilitate application development, *Jadve* goes beyond simply providing an API. The central theme in its design and implementation is the seamless integration of application-specific graph objects and operations with the generic *Jadve* services. The separation of abstract objects from application objects is the most important design decision. Components (services) that operate on abstract objects can generally be protected from applications; whereas those that use application objects need to provide support for application-specific operations.

In addition to a component-based design, Java's object-oriented, architecture neutral, and dynamic features are critical to ensure the extensibility, portability and customizability of *Jadve*. The three example applications demonstrated that *Jadve* can indeed be a framework to develop a wide varieties of data visualization systems.

6.1 Future Work

We plan to adopt a process-centered approach to support multi-user collaboration. The *Jadve* server can be integrated with a process engine (a workflow management component) that supports project-specific process (workflow steps) definitions. In such an architecture, the *Jadve* server is still responsible for supporting group communications. However it needs to coordinate group activities according to the process states managed by the process engine. For example, when a *Jadve* client sends a “review” request to the server, the *Jadve* server passes this message to the process engine, which in turn enters the initial process state of a review session. Thereafter, as the session progresses, the process engine gets new process data from the clients (through the *Jadve* server), and moves through the next logical process steps. The *Jadve* in turn instructs the clients the correct actions to take according to these steps. The advantages of using a process engine is separation-of-concern: *Jadve* can provide the generic support (i.e., group membership, message passing, concurrency control, etc.) that is required for all collaborative activities; the process engine is responsible to ensure that the activities are carried out according to the defined (process) semantics.

It is our belief that different frameworks should complement each other in terms of the services they provide. Thus an interesting extension to *Jadve* is to integrate it with other existing frameworks and provide data visualization services. PCTE (Portable Common Tool Environment) [LM93] is a framework to facilitate the development of software development environments (SDEs). It has three main sets of services: user interfaces, communications and object storage and management. PCTE provides Motif-compliant tools for GUI development, message passing and notification for CASE tools communications, and a standard objectbase for all artifacts of a SDE. *Jadve* can be added to the PCTE framework to provide visualization services, for example, to display and analyze the object hierarchy of its objectbase and the relations among the CASE tools.

7 Acknowledgment

The authors would like to thank Eleftherios (Lefty) Koutsofios, Stephen North, John Mocenigo, and Robin Chen of AT&T Labs - Research, and Jeff Korn of Princeton University for their helpful discussions.

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* Addison Wesley, Reading, MA, 1996.

[BKC95] Naser S. Barghouti, Eleftherios Koutsofios, and Edith Cohen. Improvise: An interactive, multimedia process visualization environment. In *Proc. of the 5th European Software Engineering Conference, ESEC '95*, Barcelona, Spain, September 1995. Springer-Verlag.

[CFKW95] Yih-Farn Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A graphical navigator for software and document repositories. In *International Conference on Software Maintenance*, Nice, France, October 1995.

[FPSS96] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. The KDD process for extracting useful knowledge from of data. *Communications of the ACM*, 39(11):27–34, November 1996.

[Joh93] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings of Conference on Objected-Oriented Programming, Systems, and Applications*, pages 63–76. ACM, 1993.

[KD91] E. Koutsofios and D. Dobkin. Lefty: A two-view editor for technical pictures. In *Graphics Interface '91*, pages 68–76, Calgary, Alberta, 1991.

[KK96] Jeffrey Korn and Eleftherios Koutsofios. Chava: Ciao for Java. In *(WWW pages) http://www.cs.princeton.edu/j̃lk/chava*, 1996.

[KN93] Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot.* AT&T Bell Laboratories, October 1993.

[KN94] Eleftherios Koutsofios and Stephen C. North. Applications of graph visualization. In *Graphics Interface '94*, pages 235–245, 1994.

[LM93] Fred Long and Ed Morris. An overview of PCTE: A basis for a portable common tool environment. Technical Report CMU/SEI-93-TR-1, ESC-TR-93-175, Software Engineering Institute, Carnegie Mellon University, March 1993.

[New88] F. J. Newbery. EDGE: An extendible directed graph editor. Technical Report 8/88, University of Karlsruhe, Institute for Informatics, June 1988.

[SPT+97] S. Stolfo, A. Prodromidis, S. Tselepis, W. Lee, D. Fan, and P. Chan. Jam: Java agents for meta-learning over distributed databases. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, Newport Beach, CA, August 1997. AAAI Press.