Grappa: a GRAPh PAckage in Java

Naser S. Barghouti*
Bear, Stearns & Co., Inc.
New York, NY, USA
naser@bear.com

John M. Mocenigo
AT&T Laboratories
Florham Park, NJ, USA
john@research.att.com

Wenke Lee†
Columbia University
New York, NY, USA
wenke@cs.columbia.edu

Abstract

Grappa is an extensible graph drawing package written in Java. The package comprises classes that implement graph representation, presentation and layout services. It provides an application programming interface (API) on top of which Web-based applications that need to visualize information in terms of graphs, such as process flows, business workflows or program dependencies, can be built. Through subclassing, the classes that implement an application inherit graph drawing and layout services provided by *Grappa*; these services can be enhanced and customized for the specific application. To illustrate its utility, a new version of Improvise, a multimedia process modeling environment, was written on top of *Grappa*.

---

*This author's contributions were made while employed at AT&T Laboratories
†This author's contributions were made during a summer internship at AT&T Laboratories

1 Introduction

In many domains, information can be organized and visualized in terms of graphs. Examples include process flow diagrams, software architectures, business workflows, data dependencies in data mining, and routing information in network management. Many applications supporting these domains include graph drawing modules. Not surprisingly then, there has been a significant amount of research on how to store, layout and display graphs most effectively.

With the increasing importance of Web-based, and, more generally, distributed computing, there has been an interest in providing packages for remote graph drawing services. By remote graph drawing we mean the ability for an application running on one machine to request graph drawing and layout services from a server running on another machine anywhere on the network. North's email-based graph service is an example of a remote graph layout service: it receives a graph specification as input via email and returns, also via email, a graph drawing whose layout is computed by *dot* [3]. The graph server at Brown University [2] is more extensive in that it provides interactive graph drawing and translation (among many graph drawing algorithms) via the WWW.

This paper presents a graph drawing package, *Grappa*, on top of which Web-based applications requiring graph drawing services can be built. The Grappa approach differs from the Brown graph server approach in that it provides an Application Programming Interface (API) so that applications can be developed to not only use its standard services, but also extend and customize them to accomplish application-specific tasks. This alleviates the burden of writing all the code that manages the graph drawing aspects of an application. The application writer need only concentrate on the semantics of the application and how to map those semantics onto graphs.

The advent of Java [1] motivated us to exploit the features it provides, such as portability, graphical user interface generation facilities, and distributed computing features. Therefore, we built *Grappa* in Java, which allows *Grappa*-based applications to 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.

Other client-server applications (not necessarily over the WWW) can also be built on top of *Grappa*. In this case, a *Grappa* server provides a set of standard graph drawing, editing and viewing functions; the front-end client 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 representations can be stored by the server and shared by multiple clients.

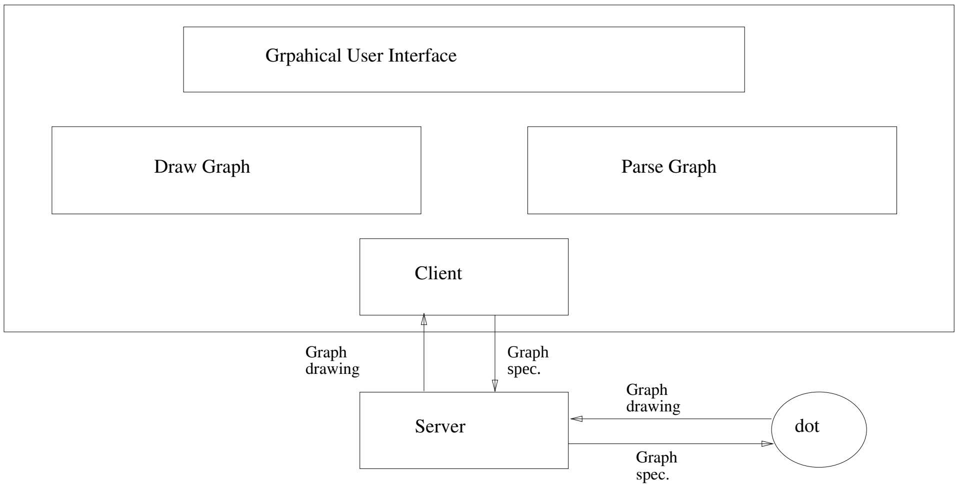

Figure 1: The Architecture of *Grappa*

2 The Design of *Grappa*

We had three main requirements in building *Grappa*:

1. Extensibility, which allows for a natural evolution path, where new services can be incorporated into the package.
2. Portability, which is essential in the multi-platform world in which we live.
3. Customizability, which allows for different kinds of application to be built on top of *Grappa*, re-using its facilities and avoiding re-writing a lot of code.

We discuss how these requirements were met in the design and implementation of *Grappa*.

2.1 The Components of *Grappa*

Grappa is a class package written in Java. The classes implement a client-server architecture that is inherited by applications. Figure 1 shows the main architectural components of *Grappa*. The server processes client requests (received in the form of messages) for graph layout. The server creates a Java thread to process each client message. The header of each message specifies the request type, and the body contains a graph specification (in *dot* format). The server invokes *dot* to compute a graph layout (drawing). The server then sends *dot*'s output, a graph drawing (also in *dot* format), back to the client.

Client classes handle most of the interactions with the end user. Its graphical user interface is constructed using the standard Java Abstract Windowing Toolkit (AWT), with pull-down menus, and canvasses for graph objects. The contents of the menus and the functions to perform menu operations can be changed for specific applications.

The *Grappa* client classes are organized in three hierarchies, as shown in Figure 2. The first hierarchy, whose root is the class *DotGraph*, contains classes for graph *definition* (i.e., for defining graphs, subgraphs, nodes, edges, and associating attributes with nodes and edges). *DotGraph* defines a graph as a set of *DotElement*s, each of which is a node, an edge, or a subgraph. Each instance of *DotElement* has set of attributes associated with it, such as *shape*, *style*, *color*, and so on. *DotGraph* also refers to the class *DrawPane*, whose instances contain information about displayed instances of *DotGraph*.

The second hierarchy, whose root is the abstract class *DrawObject*, includes classes for graph *drawing*. Class definitions for drawing 36 node shapes (Box, Circle, Ellipse, etc.) and several edge types are included in this hierarchy. Each instance of *DrawObject* refers to a specific *GraphicContext*, which provides information for drawing an object on specific canvas.

2.2 The *Grappa* API

The third hierarchy, rooted at *AppObject*, is a place holder for application-specific classes, and it constitutes the application program interface (API) for *Grappa*. An instance of *AppObject* refers to an instance of *DotElement* and has a reference to an instance of *DrawObject*; depending on the value of the type attribute of the object (e.g., the node shape), the methods of the appropriate subclass of *DrawObject* are called to draw the object on the client canvas. Application programmers can extend and customize *Grappa* by following the same approach of extending standard Java APIs. In particular, application-specific classes should be added as subclasses of *AppObject*. Application-specific behavior can then be implemented by adding data members and methods to these subclasses, overriding existing behavior.

Grappa application objects may have different drawing styles and operations for different types of application-specific information entities. For example, a program structure application may define a class *Module* as a subclass of *AppObject* and assign the value “Ellipse” to its *shape* attribute (which is automatically defined because every *AppObject* refers to a *DotElement*). The class *Module* may then define new attributes, such as *name*, *description* and *owner*, that are specific to software modules. Finally, it may specialize the *draw* method

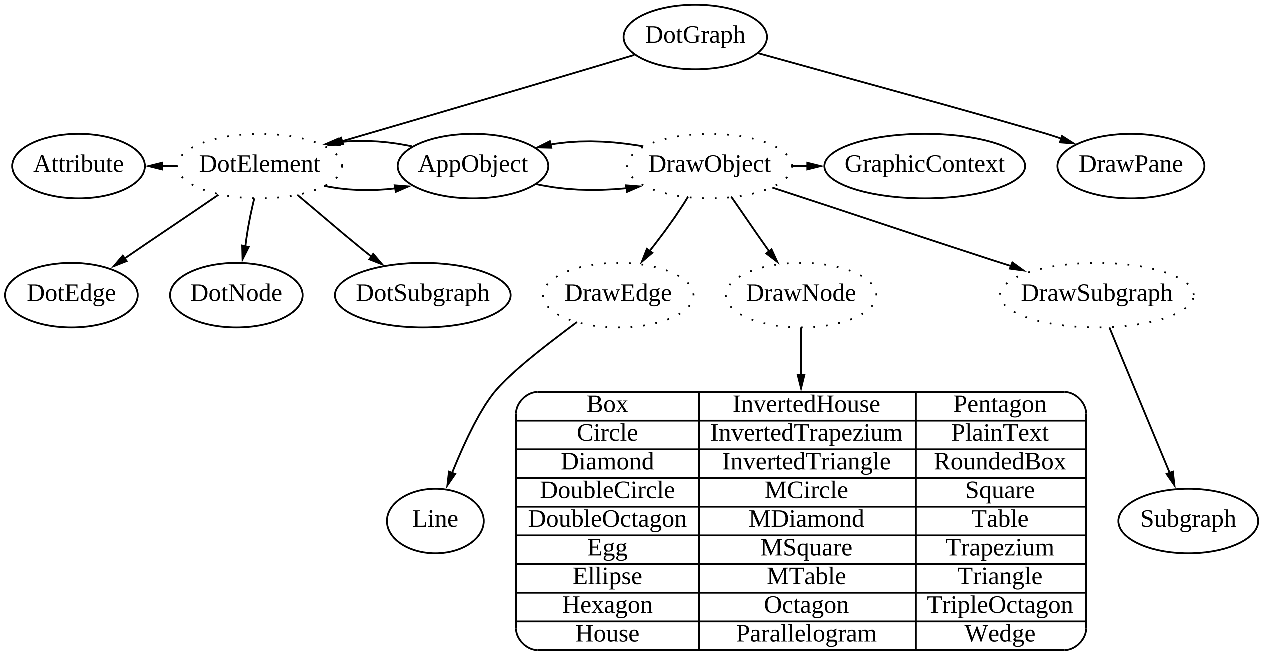

Figure 2: Class Hierarchy of *Grappa* Application Objects

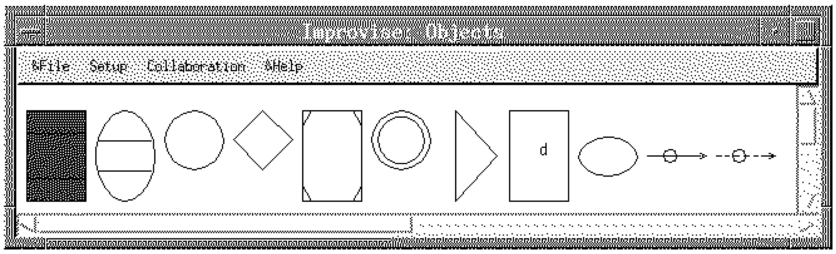

Figure 3: *Improvise* Template Objects

of class *Ellipse* (which is a subclass of *DrawNode*) to draw two horizontal lines to divide up the ellipse into three sections, and display the module's *name*, *description*, and *owner* in the three sections respectively.

3 Example: Improvise

Improvise is a multimedia process (workflow) modeling and analysis tool. It represents a process (a partially-ordered set of activities) is terms of a graph. It was first implemented on top of the graph editor *dotty* [4]. Improvise provides a graphical modeling notation that defines a set of object (node) types that represent various entities in a workflow, such as *ManualTask*, *Data*, and *AbstractProcess*; two edge types are also defined. These object types are displayed in an object palette. The user selects the type of object from the palette and clicks on an un-occupied area of a drawing canvas to create an instance of the type. An edge is drawn by clicking on an existing node and dragging the mouse to another node. Repeated selection and clicking (without worrying about placement of nodes and edges) results in a process flow diagram; on demand, the diagram is sent to *dot* to compute a graph drawing that is then displayed on the canvas, replacing the previous diagram.

Last year, we made a decision to re-implement Improvise in Java and make it executable (in a client-server manner) over the WWW. One alternative was to re-code Improvise in Java from scratch. The second alternative was to build it on top of *Grappa*, which was being designed at that time. We opted for the second option because it would relieve us from writing all the graph drawing and layout code.

Compared with using *dotty*, our experience shows that the *Grappa* API is more elegant because of four reasons: 1) the graph object hierarchy is intuitive and easy to extend; 2) *Grappa* hides the implementation details of graph drawing and layout, which are not the main concern of Improvise; 3) the dynamic features of Java enable *Grappa* to evolve in parallel and independently with application development (i.e., the layout algorithm can be changed without affecting Impro-

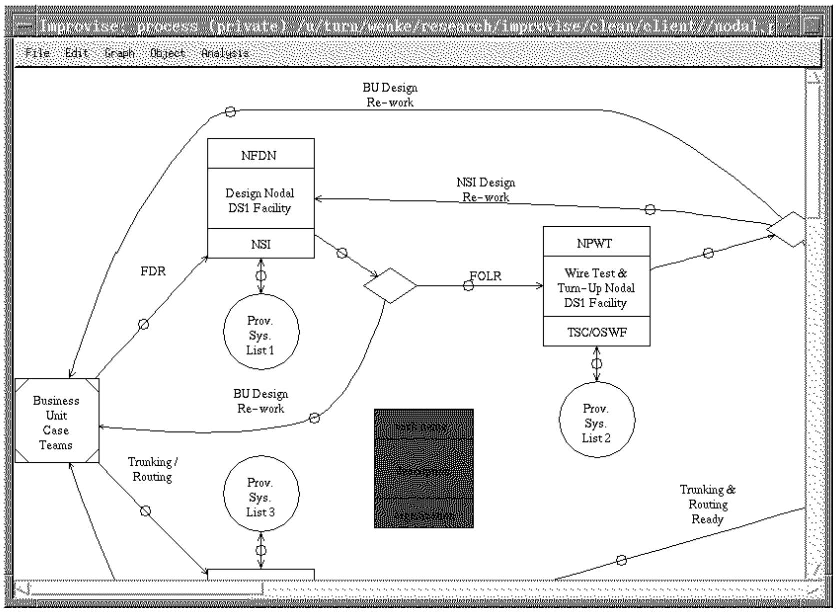

Figure 4: *Improvise* Menu and Canvas Showing a Workflow

vise); and 4) being web-based, the new version of Improvise does not require any software distribution, and new versions are immediately available.

4 Conclusion

We presented *Grappa*, an extensible graph drawing package written in Java. *Grappa* comprises a set of classes that implement graph representation and presentation services. It also provides an API for applications that require graph drawing, editing and browsing services. Application classes are defined as subclasses of *Grappa* classes; the subclasses can specialize the base classes to support application-specific behavior.

One of the motivations of our work was to explore and experiment with Java. Although still very young and evolving, Java has a lot of advanced features that will make it a success. As discussed throughout this paper, Java's object-oriented, multi-threading, and dynamic features are most important to the design of *Grappa*. Its exception handling feature can also be used as a very useful programming aid: an error stack trace which includes file names and line numbers can be printed out when an exception is raised, so that bugs can be easily located and fixed. A worth noting phenomenon that have fueled the rapid adoption of Java is that a lot of Java enthusiasts have made their programs freely available on the Internet. A lot of these programs are not just "cool", but they actually provide needed features that are not yet supported by the standard Java packages. For example, our *Grappa* parser was constructed using *JavaCup*, a $LALR(1)$ parser generator developed by Scott Hudson in Georgia Institute of Technology.

Our overall experiences with Java is quite positive. We were able to implement *Grappa* in about 10k lines of Java code in two and a half months. *dot* was not ported to Java; rather, we use it as a *native* program (in the native host environment) that is invoked from the *Grappa* server.

5 Acknowledgment

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

References

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

[2] Stina Bridgeman, Ashim Garg, and Roberto Tamassia. A Graph Drawing and Translation Service on the WWW. In *Proc. of Symposium on Graph Drawing GD'96*, Berkeley, CA, USA, September 1999. The URL is http://loki.cs.brown.edu:8081/graphserver/home.html.

[3] Emden Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, 19(3), March 1993.

[4] Stephen C. North and Eleftherios Koutsofios. Applications of Graph Visualization. In *Proc. of Graphics Interface '94*, pages 235–245, Banff, Alberta, 1994.