Transforming Command-Line Driven Systems to Web Applications

Constantinos Phanouriou and Marc Abrams
Department of Computer Science, Virginia Tech,
Blacksburg, VA 24061-0106
+1 540 231 6931
{phanouri, abrams}@cs.vt.edu

Abstract

A method to add a Web-based interface to a command-line driven system is presented. Without programming, *Javamatic* can generate a graphical user interface, then invokes commands in the legacy system transparently to the user. The user interface (UI) is automatically generated as a Java applet or stand-alone interface from a high-level description of the application, which is UI independent, using a set of UI mapping rules. The application is wrapped with an interface server; thus multiple clients can use the legacy application through the Web.

Keywords

Web applications, Java programming, legacy code

Introduction

There are many useful programs designed years ago to run on stand-alone computers that are too costly to rewrite as Web applications. In this paper we describe a method to add a Web-based interface to a command-line driven application without programming. The method uses a high level description of an application to automatically generate a user interface (UI), and then invokes commands in the legacy application transparently. It does not require any changes to the application code, nor does it require application recompilation with special toolkits. The application can be written in any programming language (compiled or interpreted) as long as the needed functionality is accessible from the command-line. Javamatic is a tool[4] that uses this method to create UIs in Java[3].

Web applications are applications designed to allow any authorized user with a WWW browser and an Internet connection to interact with them. Javamatic can transform legacy command-line driven applications to Web applications. The legacy code sits on a remote machine and is wrapped with a Javamatic interface server. The user interface is generated on the fly by a Javamatic interface client on the user's WWW browser.

Javamatic can add a modern graphical UI to legacy applications, can make them accessible on platforms to which the code has not been ported (e.g., scientific codes on supercomputers can be run from personal computers), can make them Web accessible through regular Web pages, and can permit collaboration between geographically separate users, because they share a single program and its associated data. Javamatic can be used with legacy systems that have a variety of software architectures; three are illustrated here: a monolithic FORTRAN program used in aircraft design, a collection of small programs in C++ and other languages for visualization and statistical analysis, and commands from the UNIX operating system.

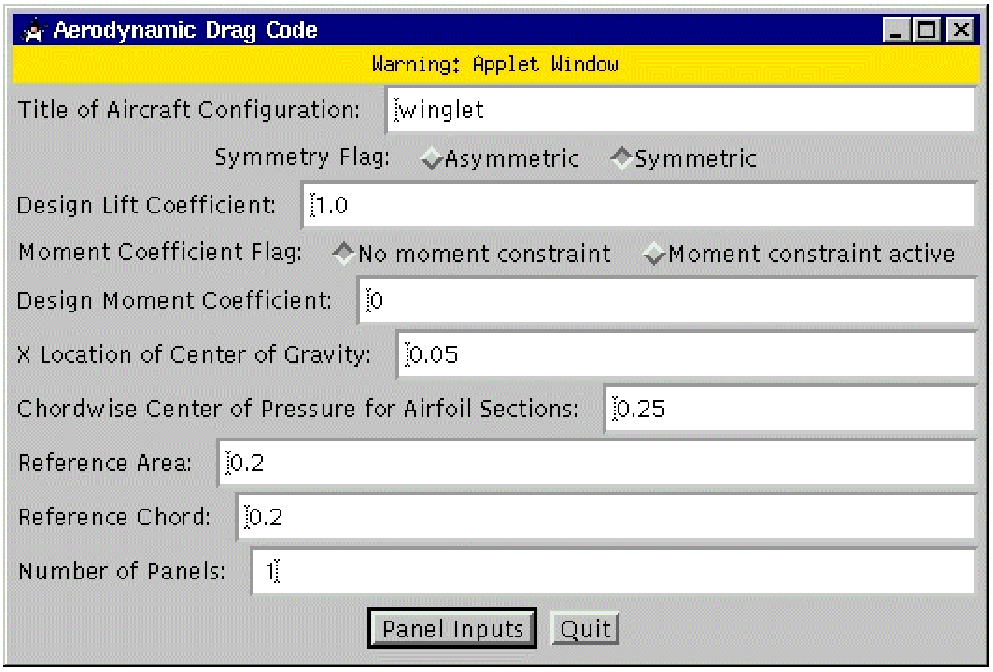
First Example: Induced Drag Program

IDrag is a monolithic FORTRAN77 program developed in the Aerospace Engineering Department at Virginia Tech to calculate the load distribution corresponding to the minimum induced drag of a given aircraft configuration. IDrag also calculates the induced drag and the span efficiency factor.

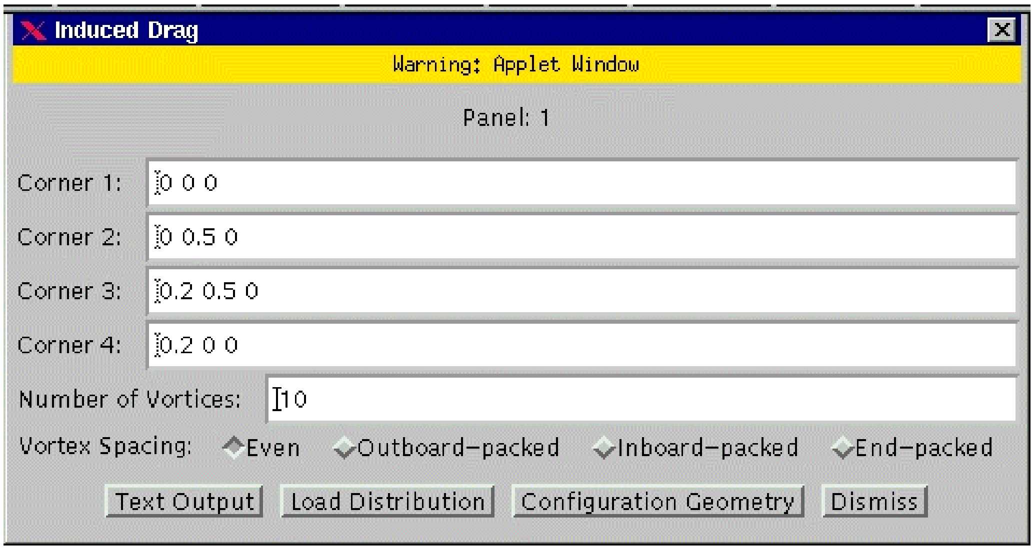
To use IDrag before Javamatic, aerospace engineering staff had to log onto a UNIX computer on which IDrag was installed, then text edit a cryptic input file containing all input parameters, listed one per line in which order is important. The user then had to invoke IDrag, which read the input file and stored the results of the calculation to another text file. To graphically visualize the results, the user then loaded the resulting text file into Matlab. During program execution the user could not interact with the program. A user could only change input parameters by editing the input file and re-executing the program.

Javamatic changed all this. Today, anyone worldwide can visit a Web page (linked from [4]), click on a link to start the Javamatic interface to IDrag, and see a modern graphical UI (GUI) to the program (Fig. 1). The program is now accessible from non-UNIX computers, even though IDrag itself was never ported to non-UNIX computers. A user no longer needs an account on a UNIX machine containing the IDrag program, no longer needs to understand IDrag's input format (due to the fields presented in Fig. 1), and no longer needs to know how to use Matlab (because Javamatic transparently invokes Matlab and generates the GIF image in Fig. 1).

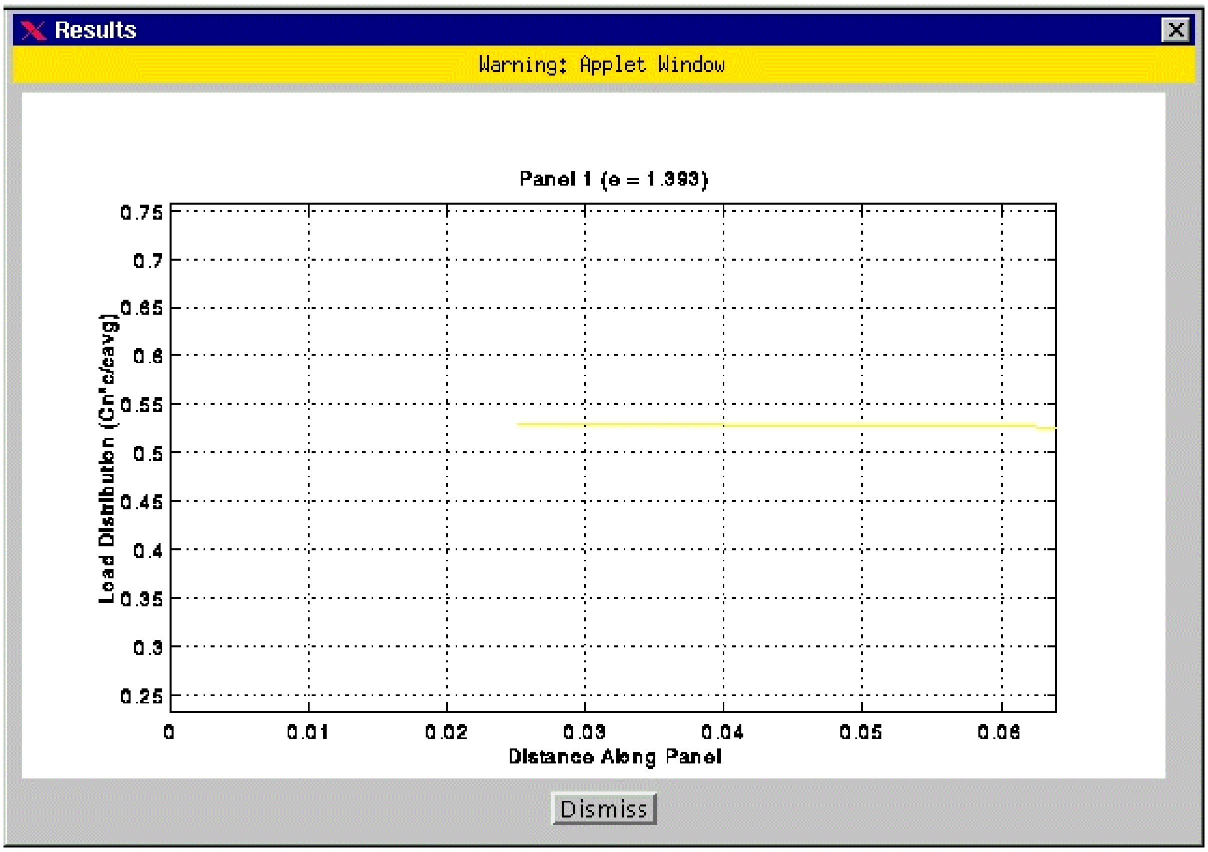
When a user visits a Web page referencing the IDrag URL (e.g., Fig. 2) in a Java-capable browser, the window in Fig. 1a appears, listing all the input parameters. The user can either accept the default values or enter new values. When done, the user clicks the "*Panel Inputs*" button and is presented with a second screen (Fig. 1b). This screen contains additional parameters that depend on the previous entries (i.e., number of panels). Finally, the user clicks on one of the three buttons (*Text Output*, *Load Distribution*, or *Configuration Geometry*) that invoke IDrag and display various forms of output. For example, clicking on *Load Distribution* shows the graph in Fig. 1c as a GIF image. The user can change any parameter in Fig.1a, which is displayed continuously, and compare old and new results. In addition, if the user has Matlab on the client computer (e.g., the one running the Web browser), Javamatic can be configured to spawn Matlab on the client machine for visualization and further analysis.



A: Initial screen seen by user upon invoking IDrag's Javamatic interface (from Fig. 2)



B: Panel Screen



C: Result screen

Figure 1. Screens generated by Javamatic

```
<applet code=javamatic.generator.Generator.class width=150 height=35>
   <param name=Application value=IDrag>
   <param name=ServerHost value=actor.cs.vt.edu>
   <param name=ServerPort value=12000>
</applet>
```

Figure 2. HTML code to execute the IDrag application using Javamatic

---

Javamatic Architecture

Javamatic is divided into two parts: the *interface client* and the *interface server* (Fig. 3). The interface client is downloaded over the Internet to the client machine and is responsible for presenting the actual interface to the user. The interface server resides on the legacy system and serves as a gateway between the legacy application and the Internet.

Interface Server

Javamatic wraps the legacy application with an interface server. This server is responsible for executing commands on the legacy system. Javamatic uses an HTTP server to transfer the application description and the execution results to the interface client.

Figure 3 shows the communication mechanism between the interface client and interface server. In step 1, an Internet user gets the URL for the application (e.g., Fig. 2) from the HTTP server on the legacy machine. The HTTP server returns an applet that is the Javamatic interface client along with three parameters: a Java class that describes the interface (discussed later) and the hostname and port number of the interface server wrapping the legacy application (e.g., IDrag). The interface client reads the three parameters and displays an initial window (e.g., Fig. 1b). Eventually, the user performs some action in the interface (e.g., clicking *Load Distribution* in Fig. 1b) that results in step 2 in Fig. 3: transmission of a request to execute a command. The interface client connects to the interface server and sends the command with optional parameters. The interface server then validates the command, and if it is valid, executes it (e.g., runs IDrag) locally on the remote system on which the legacy code is installed. After the command terminates, the server generates the output in the form of a file (e.g., IDrag writes an output file and invokes Matlab to generate a GIF), and puts the file in an HTTP-accessible directory, and returns the associated URL to the interface client. Finally, in step 3, the client issues a GET to the remote host to retrieve the output file, and displays the result (e.g., Fig. 1c).

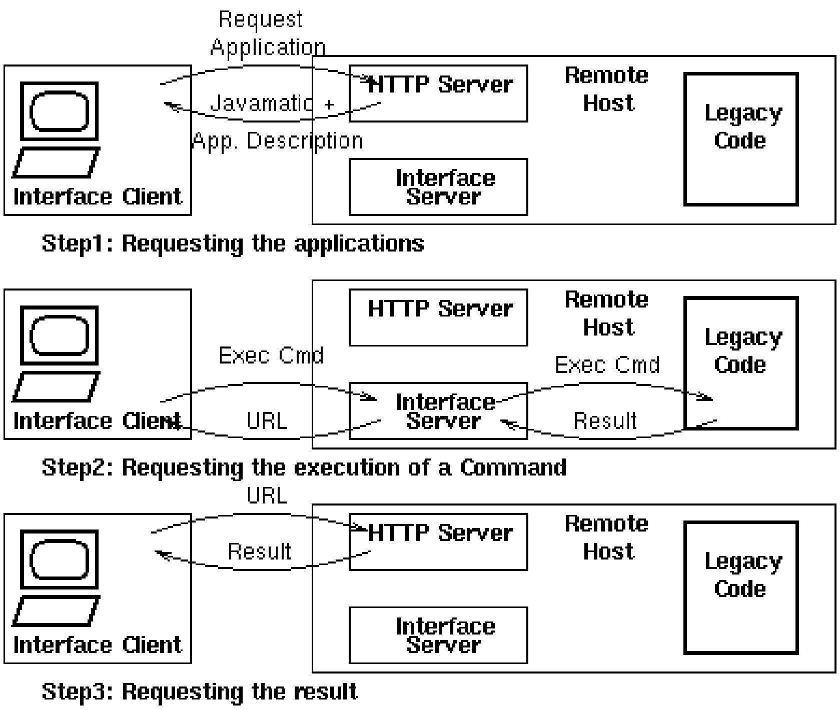


Figure 3. Client/Server Communication

The interface server can handle multiple requests from multiple clients. Each new request gets its own thread of control and is serviced in parallel with other requests. This thread is responsible for communicating the data from the client, executing the command, and doing administrative work (i.e., validating, logging, etc). If the server is overloaded, it can forward some of the requests to additional servers transparently to the client.

The execution time for each application varies. In the IDrag example above, the execution time was only a few seconds; thus Javamatic was able present the results in real time. There are many cases, however, that the execution time might be several hours or even days. In this case, the user can specify a time threshold that the server will wait for each command to execute. If the command does not terminate by the threshold, the server will continue the execution but return a URL (that will later contain the output) to the client immediately. The user can load the URL at a later time when the result is available.

To ensure that users do not execute arbitrary commands on the legacy system, in step 2 the interface server uses a *dictionary* file to protect the remote host. The client sends a command identifier, and the server queries the dictionary file for the actual command name and its path. If the command is not found, then the server responses with an error code and terminates the connection. The server also checks the command line arguments for illegal characters. If any argument fails the test, then the command is not executed.

The server also keeps a detailed log of all its activities. The log file helps application programmers identify the most frequently used components of their programs, and thus provides clues for future improvements. A log can also help the designers or owners of the legacy code understand what features users use.

All input data for the legacy application must physically reside on the legacy system prior to the execution of the commands. Since users are typically in a different geographical locations, Javamatic provides several ways to upload the data to the server. One simple way is to allow users to upload their data on the legacy system using ftp or file upload (HTML3.2 Forms). Files can be uploaded either to a public directory (e.g., to allow collaboration) or to a private, password protected directory. A second way to upload data is possible if the interface client runs an HTTP server: the client can pass a URL for the data to the remote host.

Once uploaded, the files are accessible from the application and any Internet user can use them. Users can also upload their data in password protected directories, thus ensuring their privacy. Javamatic also allows parameters to be specified as URLs instead of actual data. Javamatic transparently retrieves the URLs and presents them to the legacy application as local data. Users can store their data in one place and operate on them from different application servers without moving them around.

Interface Client

The interface client generates the GUI from an *application description* and a set of *UI mapping rules,* generated by a person familiar with how to run the application (Fig. 4).

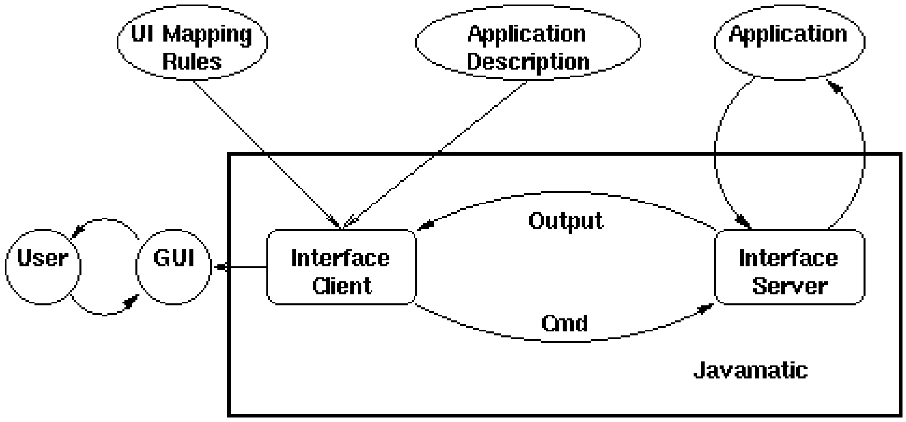


Figure 4. Javamatic Architecture

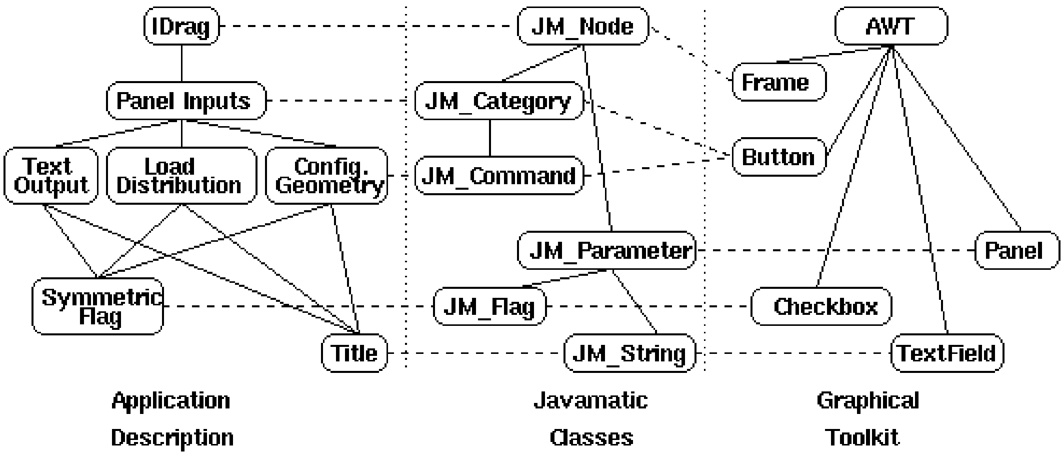


Figure 5. Creation of the IDrag interface. Dashed lines denote UI mapping rules.

Application Description

Javamatic views an application as a collection of commands (e.g., *Text Output, Load Distribution, and Configuration Geometry* in IDrag). These commands can be grouped into logical categories (e.g., two for Javamatic, corresponding to Figures 1a and 1b). These categories can themselves be grouped into more general categories until a single category is formed, the application (e.g, IDrag). The hierarchy of commands and categories can be visualized as a hierarchical tree with the application at the root, the logical categories at the nodes, and the commands at the leafs (left part of Fig. 5). Each node in the tree contains information for that category or command.

The description of an application is captured in a hierarchy of Java classes, which either an application programmer generates by writing a set of Java classes, or by a non-programmer, by dragging and dropping icons in a graphical editor (which then generates Java classes). These Java classes (on the left of Fig. 5) are derived (or *extended*) from Javamatic library classes (in center of Fig. 5). Categories are extended from the `JM_Category` class and commands from the `JM_Command` class. `JM_Node` is an *abstract* class that serves as the root of all Javamatic classes and provides the methods to include information about the application.

Both commands and categories can take any number of parameters, which are derived from Javamatic class `JM_Parameter`. For example, in Fig. 1a an IDrag user enters a string title in the first field ("Title of Aircraft Configuration"), then chooses radio buttons marked "Symmetric Flag." These appear as leaves "Title" and "Symmetric Flag" on the left of Fig. 5, and are mapped to Javamatic classes `JM_String` and `JM_Flag`, which represent any type of string and Boolean flag, respectively. Other application parameters and Javamatic classes are not shown in Fig. 5.

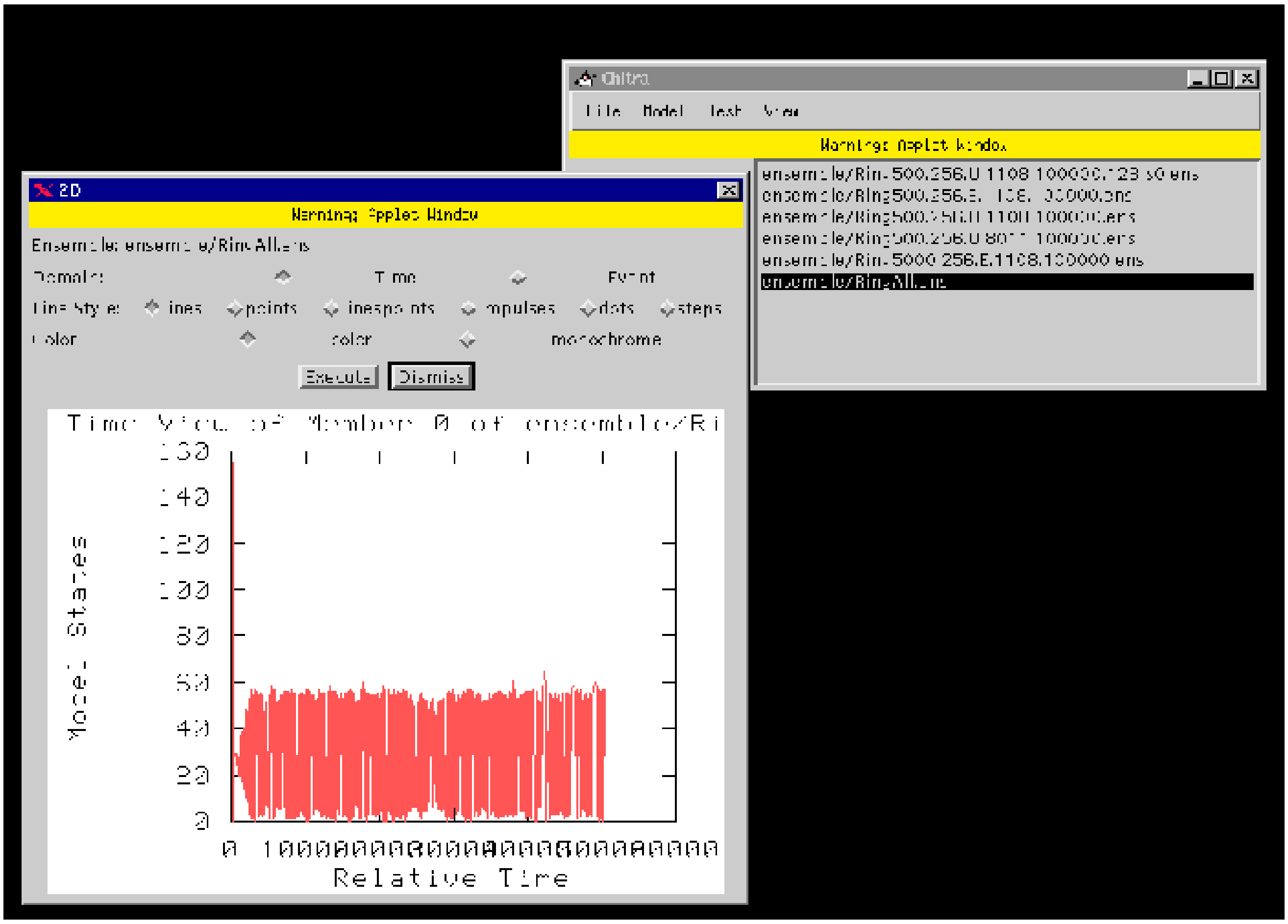
UI Mapping Rules

So far, in the left and middle trees of Fig. 5, we have enumerated the features that should appear in the interface and their mapping to Javamatic classes. However, we must next specify the mapping from of those features to GUI components (dialog boxes, buttons, etc.) This is done by specifying *UI mapping rules*, which are the dotted lines from the Javamatic classes to classes in a graphical toolkit, such as the Abstract Windowing Toolkit (AWT) or the Graphical Java Toolkit (GJT) [2]. For IDrag, everything that is a category (e.g., nodes *IDrag* and *Panel* in the left of Fig. 5) is each mapped to a separate `Frame` in the AWT (producing the two windows in Fig. 1a and 1b). The "Symmetric Flag" field in Fig. 1 is rendered as a checkbox in the GUI because of the dotted lines from class `Symmetric Flag` to class `JM_Flag` to AWT class `java.awt.Checkbox` in Fig. 5.

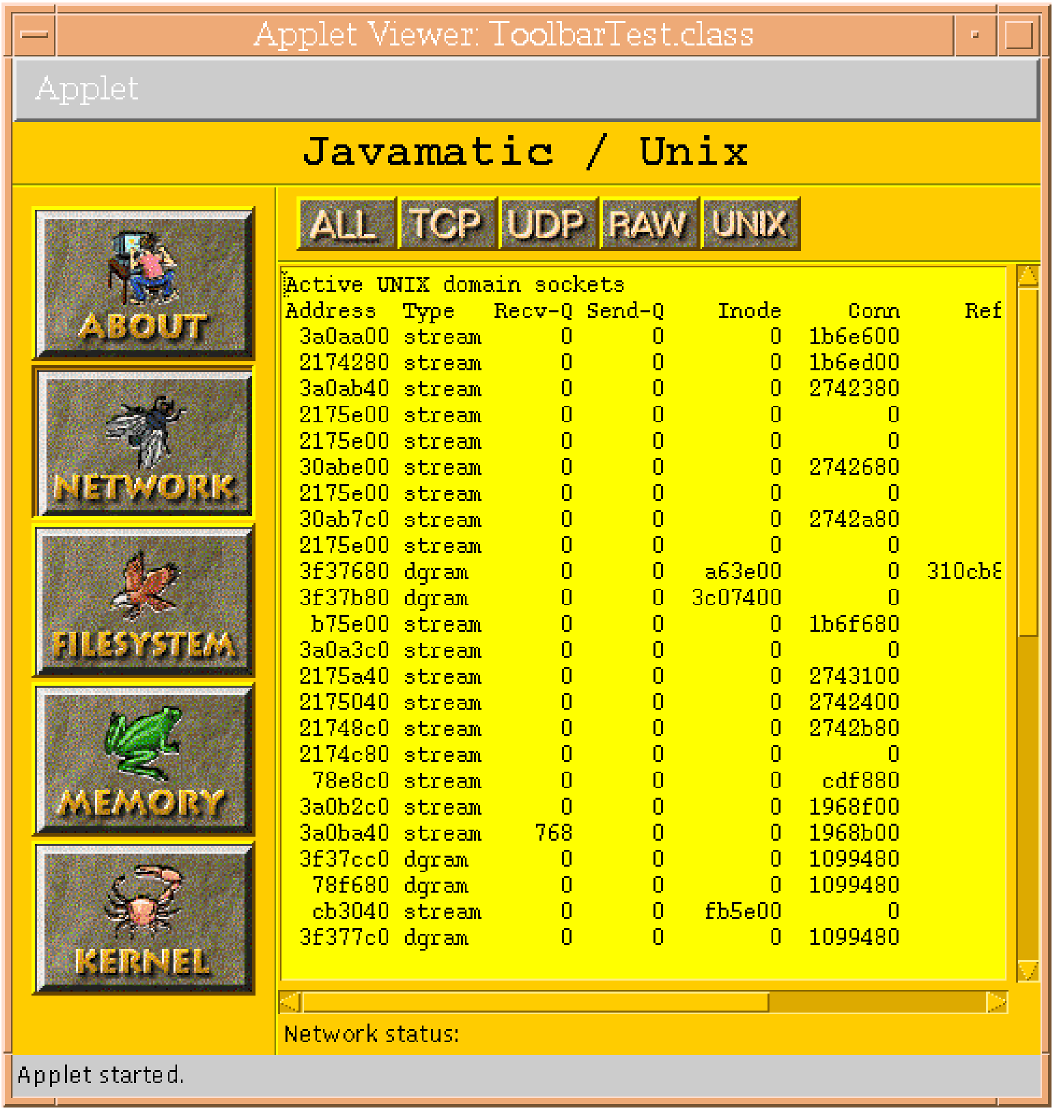
Controlling the GUI Look and Feel

By changing the UI mapping rules (e.g., the dashed lines in Fig. 5), one can easily change the appearance of a Javamatic generated GUI. For example, we could change IDrag from using two windows for input and buttons to invoke commands to a single window for all input with a menu bar whose menu items invoke commands as follows: Change the dashed line from `JM_Category` to AWT class `java.awt.Menu` and change the dotted line from `JM_Command` to `java.awt.MenuItem`.

In fact, a variety of interface styles can be created. Fig. 6 shows two different examples of using Javamatic (links to these are also contained in [2]). Figure 6a shows a menu-based interface to Chitra [1], a system to visualize, statistically analyze, and model what is known in different fields as trace files, log data, or time series data. Figure 6b also shows that the window containing the GIF image also contains a set of controls for things like image color, axis range, and so on. Javamatic classes permit attributes to be set to control button size and to allow attachment of images to buttons, which are used in the UNIX interface. The interface in Figure 6b, unlike those in Figures 1 or 6a, does *not* open new windows on the user’s screen - the interface contains exactly one window, which is inserted in the Web page that invokes the applet.



A: Menu-based Chitra interface (click image to enlarge)



B: Icon-based interface for some UNIX commands

Figure 6: Different looks and feels in Javamatic interfaces

Conclusions

The three examples in this paper -- IDrag, Chitra, and UNIX -- show the variety of systems for which Javamatic can create an interface. IDrag is a single, monolithic FORTRAN program. In contrast, Chitra is not a single program, but rather many small, self-contained programs written using C++, UNIX-shells, and Perl, each of which are command-line driven. Thus Javamatic puts a common, uniform interface to these many programs, so that the user is unaware that the underlying code was written by many different individuals. In fact, whenever someone writes a new program to add to Chitra, by simply adding nodes to the Chitra application description tree (analogous to the left side of Fig. 5), the interface for the new program appears with a consistent appearance to all other programs in Chitra. Javamatic also solved the problem of porting Chitra's code in multiple languages to different platforms; with Javamatic, the system can now be run from any Java-capable browser on any platform. The third example, UNIX commands, shows that adding a Javamatic interface requires no modification of the underlying source code. The interface in the right of Fig. 6 simply invokes Unix System V commands on an unmodified UNIX workstation.

Work is under way to attack three new problems with Javamatic:

Collaboration:
: Because Javamatic potentially synthesizes all interface code it is possible to incorporate collaborative features, so that users in remote locations can interact with each other. For example, in a classroom environment, a teacher could watch each student's interactions with a Javamatic-based Web application. Also, two users could jointly use the same server and communicate by written messages, audio, and video. Active groups of collaborators on the Web could be displayed to permit others to join the groups.

Transparent load balancing at server:
: Javamatic hides the execution of the legacy code from the user, thus the user sees the application as a black-box. The legacy code can be executed on a single machine, or distributed among different machines on the Internet.

Customizing the GUI by example:
: Although automation can generate correct and usable interfaces, it cannot compare to human creativity in generating smart and elegant designs. Javamatic can provide an initial design that users can modify. If a user modifies one window in the interface, Javamatic would infer how to update the rest of the interface from this example.

Acknowledgments

We thank Joel Grasmeyer for providing the IDrag simulation. This work was partially supported by NSF NCR-9627922.

References

1. M. Abrams, A. Batongbacal, R. Ribler, D. Vazirani, *Chitra94: A Tool to Dynamically Characterize Ensembles of Traces for Input Data Modeling and Output Analysis*, Dept. of Computer Science, Virginia Tech.
2. D. Geary and A. McClellan, *Graphic Java: Mastering the AWT*, 1997, SunSoft Press.
3. Java, Sun Microsystems, http://www.javasoft.com/.
4. Javamatic Web Page, http://www.cs.vt.edu/~javamatic(16 December 1996).