Implementation of a Java Graphical User Interface for the Visual Programming Language Forms/3

by

David P. Hackenyos

B.S., University of Utah, 1989

August, 1998

A project report submitted to the Department of Computer Science and the Faculty of the Graduate School of Oregon State University in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Margaret Burnett, Major Professor

Dr. Gregg Rothermel, Committee Member

Dr. Toshi Minoura, Committee Member

Defended August 13, 1998

Acknowledgments

I wish to express a deep appreciation to my Major Professor, Dr. Margaret Burnett. Since we met in the winter of 1995, she has been a source of constant inspiration, energy, and guidance. Her high standards of excellence in the pursuit of visual programming language research has provided me with a sense of achievement for which I constantly strive. I am grateful for all the things Peggy has done for me during my tenure here at graduate school. I have indeed been very fortunate to have collaborated with such an outstanding personality at the cutting edge of computer science research.

I would also like to thank all the members of the Forms/3 group for their invaluable help. Special thanks to John Atwood, and Rebecca Walpole Djang for their personal attention to my constant barrage of questions. There knowledge of the Forms/3 implementation is great and they never hesitated to answer any question I had for them. I would like to thank Zachary Welch for writing the LispParser class. It was a welcome solution to a nagging problem.

I would like to thank Dr. Gregg Rothermel and Dr. Toshi Minoura for being on my committee.

Lastly, I would like to thank my wife, Kathy, whose steadfast support and encouragement are the reasons I am here today, writing this paper.

Table of Contents

<u>Abstract</u>

The visual programming language Forms/3 currently uses a graphical user interface implemented in Garnet. Garnet was developed by the User Interface Software Group in the Human Computer Interaction Institute at Carnegie Mellon University, but is no longer supported. This paper presents an implementation of a user interface for Forms/3 written in Java. In addition to the implementation it also discusses the Model/View/Controller architecture used, the asynchronous socket traffic required for communication with the underlying Lisp engine, and a few design patterns found helpful in the implementation.

1. Background

1.1 Visual Programming Languages

Visual programming languages are languages that use a visual syntax, meaning that at least some of the terminals of the language grammar are graphical, such as pictures, forms, or animations . A visual syntax implies the context of two dimensions, incorporating spatial information such as containment, connectedness, location and color. Visual languages do not exclude text since it can be used for providing formulas as well as comments and labels. Examples of languages using visual syntax include some diagrammatic languages, dataflow languages, and state-transition languages in which the nodes and arcs are terminals. Some iconic languages use position to give meaning to action rules [Burnett et al. 1995].

1.1.1 Forms/3

This paper addresses the graphical user interface for the language Forms/3. Forms/3 is a declarative, form-based visual programming language that follows the spreadsheet paradigm. Detailed descriptions of the language can be found in [Burnett 1991, Burnett and Ambler 1992, Burnett 1993, Burnett and Ambler 1994]. A programmer places cells on forms and provides formulas for these cells. A formula can be made up of constants, references to other cells or references to the cell's own formula at a previous moment in time. It is apparent, perhaps an understatement, that a graphical user interface is an important aspect of a visual language such as Forms/3. Prior to this point, the current interface was implemented in Garnet, a constraint based language itself implemented in Lisp. For reasons discussed in the following section it has become desirable to replace the Garnet interface with one written in Java.

1.2 Motivation

To date, Forms/3 uses a graphical user interface implemented in Garnet [Myers et al. 1990]. Garnet was developed by the User Interface Software Group in the Human Computer Interaction Institute at Carnegie Mellon University. It is a declarative, constraint-based prototype-instance object system development environment for Common Lisp and X11 or Macintosh [Myers et al. 1992]. At the time of Garnet's inception, most programming languages did not provide GUI tools themselves [Myers 1993]. Since Forms/3 was written in Lisp, Garnet seemed a natural fit for the user interface.

Several problems have since arisen with the Garnet interface. Garnet is no longer supported by its originators. New and helpful features are not forthcoming and current bugs are no longer being addressed. Garnet has a big space leak. That is, objects thought to have been destroyed are not being handled by the garbage collection activities of the underlying Lisp implementation. The unstructured nature of the constraint-based approach presents to the programmer some difficulty in obtaining a robust application and crashes occur more often than is desirable. Garnet is an application containing a large number of tools written in a paradigm not widely used by programmers, and separate to the underlying Lisp implementation. As such, there is a big learning curve for new members to the Forms/3 team of researchers.

One of the goals of a graphical user interface should be to obtain robustness since it must deal with a variety of user inputs [Myers 1993]. The original implementers of Forms/3 were not careful to separate the interface from the Lisp engine and much of the code is intermixed. This is not totally desirable and does detract from the robust nature of the application but at the time, fit well into the rapid prototyping scheme of "plan to throw one away".

Providing a graphical user interface in Java for Forms/3 is an attempt to address these issues. Java is a popular and supported language for writing in a purely object-oriented style with which most programmers today are familiar. The graphical tools are part of the language itself and the garbage collection activities of Java are aware of all objects, graphical or otherwise. A new implementation in Java will provide an opportunity to disentangle code related to the user interface from code in the underlying Lisp engine. This approach fits well into the Model/View/Controller pattern discussed in the next section.

In addition to this better View/Model separation, Java will allow Forms/3 to achieve better cross platform compatibility. With a front end user interface written in Java, it is conceivable that Form/3 could eventually run on any machine, regardless of the architecture or operating system as long as it was running the Java Virtual Machine. It is also conceivable that a few changes from application to applet could result in web access to the front end of Forms/3.

2. Design Decisions

Before any significant code was written a lot of time was spent considering major design principles for the project. Since this project represents the starting point of an application that may be continued far into the future by many others, it was important that the design be one that lends itself to the ideas of ease of maintenance and understanding. The code should also be easily augmented. The subsections below discuss some of the important design decisions that came from these considerations.

2.1 The Model/View/Controller Pattern

The Model/View/Controller(MVC) pattern was originally developed for the creation of user interfaces in the Smalltalk programming language [Krasner and Pope 1988]. In Smalltalk, the controller and view objects are themselves part of the development environment and integral to the language [Bourne 1992]. Simply stated, the goal of the MVC design pattern is to separate the application object (model) from the way it is represented to the user (view) from the way in which the user controls it (controller). Figure 1 illustrates the communication among these three modules.

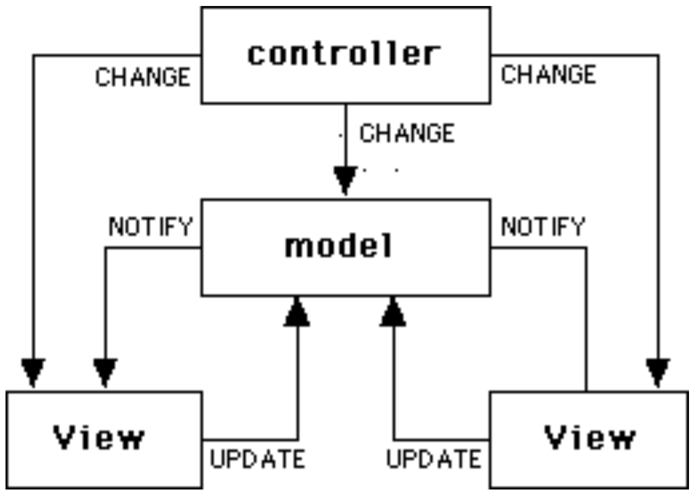


Figure 1: The classic Model/View/Controller architecture

Figure 1 also shows another important aspect of the MVC paradigm, in that more than one view may be managed by a model and controller. In some cases each view may have its own controller since the interpretation of user input through a view may tightly couple the code of a view - controller pair [Eliens 1995]. In cases where the programmer may expect the interactions to change frequently, or where one might expect many different applications to display the same data but manipulate it differently, separating the view and controller is a good idea [Martin et al. 1997].

Technically, since Forms/3 is a visual language that lives in a windowing environment that supports standard event-handling mechanisms for user input interactions (buttons, menus, etc.), the specific type of pattern this project implements is called an Event-Driven MVC Architecture [Lee 1993]. In this type of architecture, the controller component becomes the event handler that dispatches user input-device events, windowing-environment events, and other system events to the corresponding response methods of the model object.

The model object is that part of the application that deals with the information to be utilized. It typically contains the data structures to hold the data along with accessor methods by which the view may query the current state and mutator methods by which the controller may change the state of the data.

The view object is responsible for displaying the data to the screen. It is that part of the application that the user sees and with which the user interacts. When a view is asked to update itself, it queries the underlying model for information about the state and then draws itself to the display.

The controller knows about the physical means by which the users manipulate data within the model. It receives and translates mouse clicks and keyboard presses into the methods the model understands. In some situations the controller may interact directly with the view without passing through the model. This would occur, for instance, if the view was asked to display data that was merely being reformatted without any changes to underlying model.

Each model object has an "observer relationship" with its view or views. When mutator methods are called for the model, it broadcasts a "notify" message to all its associated views. This message asks each view to update itself based on the current state of the model [Collins 1995].

This project deals with creating a new view and controller for an existing model with which it must communicate over a socket (discussed below). This decoupling of the model, view, and controller aspects of an application not only allows for greater flexibility, re-use, and maintenance, but provides an easy way to implement the new GUI with the existing model. This project provides a Java model cache that merely caches the information found in the underlying Lisp implementation. Eventually, in an effort to reduce this socket traffic, the existing model may someday be replaced with a model also written in Java. This is a necessary change to the classic Model/View/Controller structure to adapt to the presence of two models of which one, the Java model cache, is an intermediary between the View/Controller objects and the Lisp model. Figure 2 illustrates this adaptation to the structure.

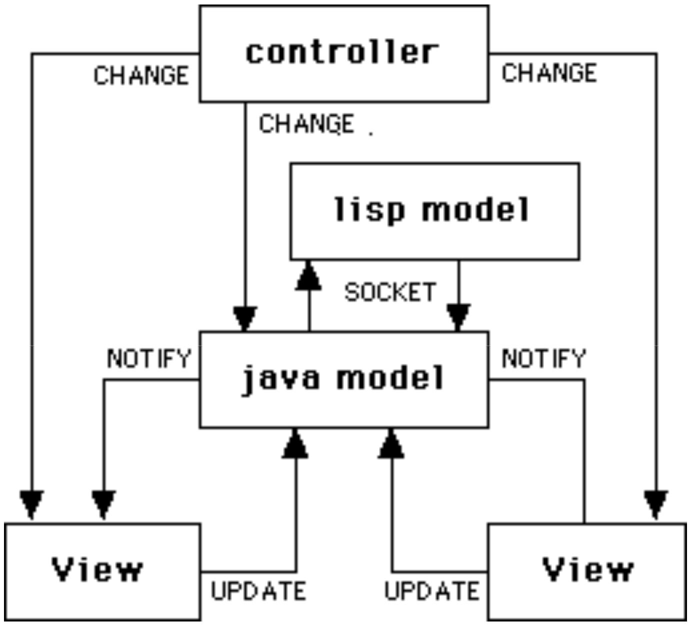


Figure 2: The MVC architecture adapted for the Java implementation

The flexibility of such a design will allow for future migration of the model from Lisp to Java, if desired.

2.2 Compatibility with the Existing Lisp Architecture

Another important design decision was to be consistent with the existing Lisp architecture. In the underlying model, there is already an established hierarchy representing the various kinds of objects with which the user may interact. The Java implementation has retained this structure, preserving the nomenclature as well. Persons continuing this implementation who might be familiar with the current way the classes are organized should have little difficulty working with the code found on the Java side. Any future changes that may occur on the Lisp side could be simultaneously implemented on the Java side. Since the Lisp model is reflected in a Java model cache, changes to the underlying model could also be made to both sides concurrently, allowing for an easy switch to a complete Java implementation in the future.

2.3 Asynchronous Socket Traffic

Because this graphical interface must communicate with the Lisp engine, a socket was chosen as a vehicle to transmit and receive messages. The mouse clicks and key presses produced by the user would be interpreted and, if needed, packaged into appropriate messages and sent to the underlying Lisp model through the socket. Once the message was processed, information is packaged into a return message and sent back to Java for display. This situation sets up a traditional Client/Server relationship, with the Java GUI containing a class to represent the Client (discussed below) and a function on the Lisp side playing the role of Server.

At first the Java implementation relied on a Client class that not only sent the message but waited for a return message from Lisp with the next line of code in a sequential manner. Soon it became apparent that some situations would not lend themselves to this communication scheme. The user may produce messages faster than the Client could receive and process the responses, such as during the wait for garbage collection by the Lisp engine. A single message from Java could generate several response messages producing a similar lag. With asynchronous communication at least the Java GUI would not be stalled. In the case of an unsolicited message sent by the Lisp side, this sequential arrangement would fail altogether.

As a result of these considerations, it was decided to incorporate an asynchronous communication structure. Two classes were designed, one to send messages, and the other to listen for them. The listening class spins off a thread whose sole purpose is to listen for messages from Lisp. These singleton objects would act independent of each other. Since the incoming message was no longer sequentially related to the message sent, some way was needed for the returned message to tell the receiving class what type of message it was and for which object the message was intended. Each message type was given a unique keyword to identify its nature and a formID and cellID with which to route it. In this way an incoming message could be processed by the Java Client class without knowing anything about how the response was generated. Section 3.2 will cover the implementation of this scheme in more detail.

2.4 Design Patterns

One of the latest developments in object-oriented design is the inclusion of design patterns. An architect named Christopher Alexander coined the idea while speaking of patterns in architectural design [Alexander et al. 1977]. The idea was originally chronicled for the software development community in the book *Design Patterns* [Gamma 1995]. It contains 23 different solutions to particular classes of problems. A pattern embodies a complete idea within a program and thus it can appear at the analysis phase or high-level design phase [Eckel 1998]. The notion of patterns provides a higher leverage form of reuse that focuses on identifying the common behavior and interactions that transcend individual objects [Booch 1996]. Some patterns are already known to most programmers of object-oriented programs, although they may not have thought of them in this way. Inheritance is a pattern (though implemented by the compiler) and composition can also be thought of as a pattern. An "iterator" or in Java an "enumeration" hides the particular implementation of the collection as one steps through and selects the elements one by one. This is also a pattern [Eckel 1998]. Patterns describe relationships between objects in a system but at a finer granularity than that of the complete application. It is the realm of design that falls between the implementation of separate objects and what is known in software development circles as a "framework" [Booch 1996].

A singleton is arguably one of the most simple of patterns. A singleton is simply one class or one instance of a class with which other class clients will make requests. Singletons provide a way to localize code for use by many others. Several situations arose in this implementation where a singleton seemed to be the proper solution. Objects involved in the socket traffic need be only singletons. The Global class is a singleton which houses the items of global interest.

Another pattern prevalent in the code is called the Chain of Responsibility pattern. Here a message is passed along a chain of objects until one is reached that knows how to deal with the message. This occurs in the Java implementation much of the time in routing the handling responsibility of user interaction to the appropriate object.

The Intelligent Children pattern is distinguished by the act of delegation. In cases where an object is composed from other objects the top level object may in turn delegate the message to its composed members to act upon. This pattern may be illustrated by an abstract class that contains an abstract method for an operation (such as drawing) and several derived classes that implement the method in ways unique to the derived class. The Dycon classes (section 3.6.2) illustrate this idea. Specifically, when a composeDycon is asked to draw itself, it merely delegates the drawing to its composed members.

The Factory pattern uses the idea of code isolation. Putting all the code in a single place that deals with the instantiation of objects provides for better encapsulation and maintenance. The ROFactory class contains methods to create ROs (Referenceable Objects). Additional RO types could be included in the future with little difficulty.

Patterns allow a programmer to appear more experienced than he or she may actually be. To use a pattern is to use the experience and insight of programmers who have thought about and solved a problem in the past. Patterns are a way of thinking about design and are not bound by rigid rules of implementation. They are merely guidelines by which one separates a part of the code that may change from a part that may not. New patterns are discovered every day and added to the growing list. Many patterns seem to be just extensions of what one may think is good object-oriented design and looking through this implementation might produce the discovery of other patterns already categorized but not mentioned here. Being able to draw upon the collected patterns of previous software designers allows for a wealth of ideas that have been tested and shown to solve problems to be applied to the situation at hand.

3. The Implementation

This project currently has approximately 10,000 lines of Java code that includes 65 classes. Originally started in version 1.1.2, there have been several upgrades to the last compiled version of 1.1.6. The work was compiled on a Sun machine using the Solaris 2.5 operating system.

To run the implementation, one must first start up the Lisp engine by typing "GarnetForms -n" on the machine called Gold. In the future the machine running Lisp will become a parameter of the Java side startup script. Then typing "(load GUIRUN)" followed by "(gui-run :port <someport number>)" will load and run the file containing code to start the socket listener on the Lisp side. On a Sun/Solaris machine one then runs the script called "javaforms" with the parameters of the port number chosen above, the letter 'T' or "nil" to pass to the keyword $JAVA, and the username of the operator. The Java implementation will then run all the pertinent Lisp files by socket communication.

3.1 The Forms3 class

The Forms3 class contains the application's main method. In Java, the syntax for this method is: public static void main(String Args[]) {...}. Typing Forms3 at the command line, using the syntax above, calls this method. In main, a splashscreen is shown for three seconds, the Forms3 class is instantiated, and the resulting main form is positioned on the screen.

The constructor for the Forms3 class itself creates all the menus, buttons, slider bar, and lays them out accordingly within the frame. This class also initializes the global FormTable and a global FormsList both of which are located in the Global class discussed below. A GUI object, instantiated from the FormsListGui class, is created to hold and view the FormsList. The Java Client side of the communication socket is created and several messages are sent to the Lisp Server to start up the underlying Lisp engine. The communication model is the subject of the next section.

3.2 View-Model Communication

The Java GUI will play the role of the view in the model/view/controller scheme and must communicate with the underlying Lisp model. But the project does not intend to stop there. At some point in the future it may be decided to translate the Lisp model to Java as well. With that in mind a cache of the Lisp model, the Java model, is included in this implementation along with the GUI.

Nothing in the Java model cache changes without first acquiring the information from the Lisp model. The information contained within the Java model cache is merely a reflection of the information found in the underlying Lisp model. This rule is suspended for four specific attributes of a GUI object. Since knowledge of the left, top, width, and height of an object is unimportant to the underlying models during a user session, these values are maintained by the GUI only until such time as saving or undisplaying a form needs to cause a flush of these values to the model side of the architecture. This flush is needed to provide Lisp with these values to save on disk or for the Java model cache to use upon redisplay of the form. Figure 3 shows the overall routing of information within the message passing scheme.

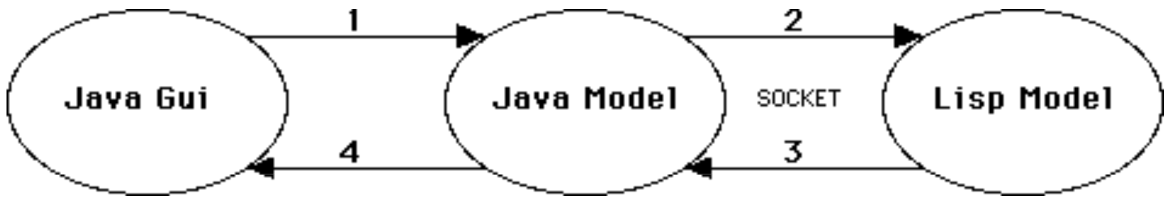


Figure 3: View/Model communication

When the user performs an action requiring communication with the model, a message is sent via the socket connection from the Java model to the underlying Lisp engine. This message has a uniform appearance in that all messages from Java begin with a "gui" function name. All the functions that begin with "gui" reside in a file called gui.lisp. The message is processed by the Lisp engine, by calling the named function, and the result is sent back across the socket. A complete listing of these messages and the format of their returned evaluations are found in the appendix.

Four classes have been implemented on the Java side to handle asynchronous communication with the Lisp engine.

- LispTalker
- LispListener
- LispParser
- MessageHandler

All four classes are singletons and are never instantiated. The methods within them are static and accessed by prepending the class name to the method call. Figure 4 illustrates the interactions among these classes.

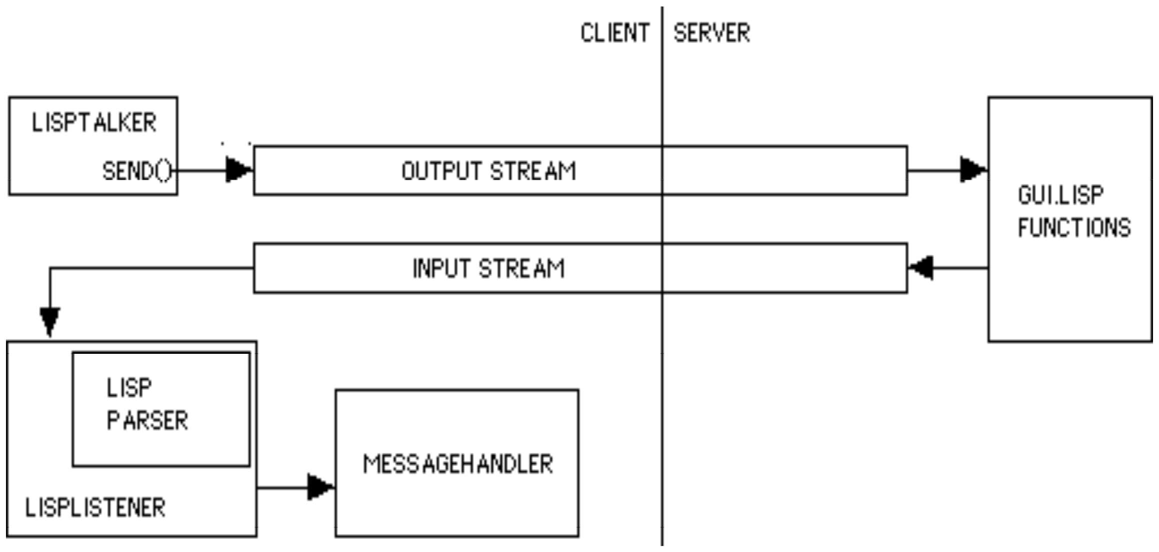


Figure 4: Classes involved in View/Model communication

The LispTalker class provides two basic functions. The first function is to establish the Client/Server relationship by creating a socket to the Lisp Server. Every socket has an input and output stream by which messages are passed from and to respectively, to the other side, and this class uses the output stream of this socket with which to send messages. The second method is to provide the functionality for printing messages to this output stream. Each message begins with a unique "gui" function name followed by the formID, possible parentID, and a cellID. Any additional information particular to the message follows.

LispListener is a subclass of thread and when asked to start by the main program, creates a thread with which to connect to the input stream of the communication socket and listen for incoming messages. Using the LispParser class, the incoming message is packaged into a string and passed to the MessageHandler class for parsing.

LispParser is responsible for capturing the entire message. Several methods within this class allow a message containing newline characters embedded within a string to be incorporated into the message instead of prematurely truncating it.

The MessageHandler class has one method whose sole purpose is to parse the incoming message. Every message begins with a unique keyword on which to parse, giving some indication of the nature of the information contained in the message. Following the keyword are the formID, a possible parentID, and a cellID to give the handler knowledge of which object to send the message. This structure of a unique keyword and a given object allows for the asynchronous reception of any message returned to the Java side of the socket.

It should be noted that this arrangement for handling the asynchrony between messages sent and those received was decided upon for flexibility. A message sent to the Lisp engine may, depending on its nature, generate more than one reply, as in the Fibinacci program returning a number of forms upon evaluating the answer formula. An incoming message may in fact have been totally unsolicited, originating on the Lisp side, such as an error message or a time dependent response unrelated to current user interaction. As long as the incoming message is packaged with the proper keyword and object information, the Java side will handle it correctly.

3.3 The Forms

Currently there are two types of forms implemented on the Java side, the SimpleForm and the VadtForm. The VadtForm is different from a

SimpleForm by having an Absbox and Image cell by which a user defined visual abstract data type may be defined. The SimpleForm is the root of an inheritance hierarchy from which the VadtForm is subclassed. As noted above, there is a class for the graphic object and a corresponding class for the Java model cache for each of these forms. The "Data" classes are part of the cached data model while classes that start with "Gui" are part of the Java view. The four form classes are:

- DataSimpleForm
- GuiSimpleForm
- DataVadtForm
- GuiVadtForm

In figure 5 below one can see the relationships among these forms.

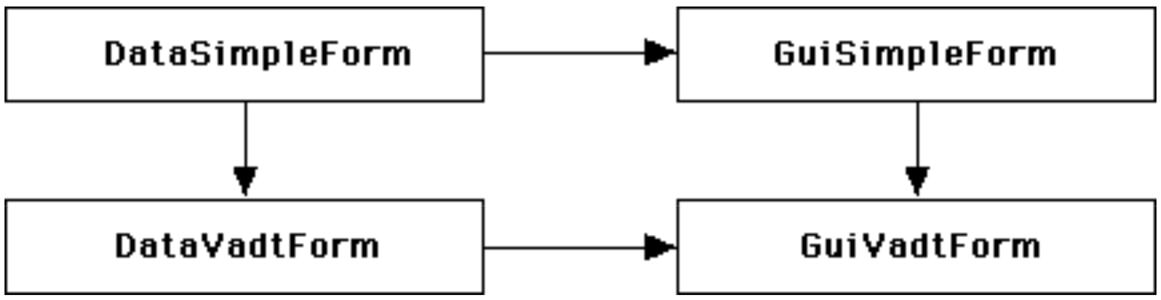


Figure 5: The two Form types, the Data(model) type and its Gui(view)

Most of the functionality for the model form is found in the DataSimpleForm class. It contains three main objects: a table of ROs (cells, absboxes, and matrices) on the form, an ROFactory to create these ROs, and a vector containing the list of Gui objects representing this data.

At the present time, the only Gui that exists is the object created by the GuiSimpleForm, but in the future it may be decided to have more than one view of the data available to the user. Thus the need for a list of Guis. Whenever information in the model changes, it is merely a matter of calling the method NotifyGui(), and if the list is empty, a gui object is created and added to the list, or if the list is not empty, views are asked to redraw themselves, updating the appearance of the Gui object to the user.

Whenever a form is created or loaded from disk, Lisp sends the appropriate information to Java via a message. After gleaning the information relative to the form itself, the remaining message is parsed in this class by the method called parseFormData(String s). Here, keywords within the message tell the model which method of the ROFactory to call to instantiate the proper type of cell.

The DataVadtForm is an extension of DataSimpleForm and adds the ability to deal with an Absbox and ImageCell indigenous to a Vadt form. Consequently, these data members and their accessors are included as well as changes made to the parseFormData method needed to produce them. The NotifyGui method instantiates a GuiVadtForm instead of a GuiSimpleForm.

The two Gui forms contain three main objects each of which provide the user a means to interact with the application: a MenuBar, a ToolBar, and the FormDrawWindow. The MenuBar provides selections allowing the user to choose types of cells to create, attributes of cells to manipulate, and the cutting of cells, as well as producing a help file for the form if one exists. The ToolBar is optional and provides buttons that mimic much of the functionality of the MenuBar. Most of the area of each form type is made up of the FormDrawWindow, in which cells are placed, resized, and moved around. It is in the FormDrawWindow that the user does his or her Forms3 programming. This important class is the topic of the next section. Figure 6 shows examples of the types of forms and their FormDrawWindows.

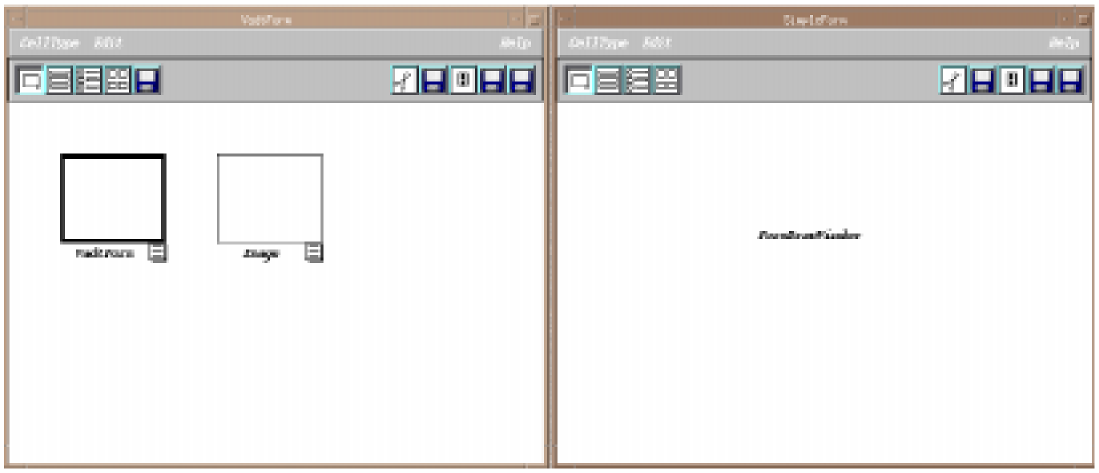


Figure 6: A VadtForm (showing the Absbox and Image cell) and a SimpleForm.

3.4 The Controller - class FormDrawWindow

As noted above the Controller for an application is responsible for handling the interaction between the end user, a Forms3 programmer in this case, and the model or its views. Mouse clicks and keystroke/mouse click combinations are translated into actions on model or view objects.

Class FormDrawWindow is the controller for this Java implementation of the Forms3 Language. All forms contain an object of this class, a double buffered container within a panel. This container uses a Layout Manager found in the gjt package, called the BulletinBoardLayout(). This customized Layout Manager allows for components to be placed freely in the container at specified coordinates relative to the upper left corner instead of being constrained to certain patterns found in other Managers. It is within the boundaries of this object that all mouse events are monitored and subsequently routed to the appropriate handlers. This class implements four methods of the MouseListener and MouseMotionListener interfaces of Java; MouseClicked, MousePressed, MouseReleased and MouseDragged. Common to all four of these methods is the acquisition of the mouse coordinates and a determination of what kind of event occurred, if the event was associated with a cell or not, what kind of cell, and where on the cell.

The MouseClicked method is the largest one and responsible for most of the interaction. Depending on what keys are pressed if any, a single click could add a cell of the type specified in the menu to the form or Absbox, add a cell reference to a formula, draw dependency arrows, select a cell for further action, dispose of a formula window, or invoke the name dialog box. A double click in a cell will invoke the cell attributes dialog box and if on a formula tab will bring forth a formula dialog box.

The MousePressed method sets the appropriate Boolean flags and coordinates of a potential rubber band, along with changing the cursor to a cross if not on a cell. If the press occurred on a cell, the cursor is changed to a moving cursor. If a press is within a formula tab, the formula window is drawn. Finally, if the press was in the OverlayRegionBar of a Matrix, a Boolean flag is set to true in preparation for possibly dragging out a new region.

The MouseReleased method will calculate the enclosing rectangle of any stretched rubber band and select the cells contained within it and resets the rubber band points. The cursor is replaced by the original. If the release was in a formula window, the window is posted in place of the formula tab. If the release was in a matrix and the bar flag was set, a region is created based on the coordinates of the mouse at the time of release.

The MouseDragged method will move one or more selected cells as a group. If the only cell selected is an Absbox, the Absbox and any cells contained within it are moved as a group.

Additional methods exist to process the result of any Dialog Boxes that the user may have invoked. These deal with giving a cell a name, formula, or attributes. Dependency arrows may be requested to be drawn or rubber banding of cells for selection may be invoked. Whenever a change does occur, the FormDrawWindow is repainted to update any change in appearance of the view.

3.5 The Referenceable Object (RO) Hierarchies

As with forms, referenceable objects (hereafter referred to as ROs) have both a data model cache, the DataRO, and its corresponding graphical object, the GuiRO. From figure 7 one can see the ancestral hierarchy of the various DataRO types.

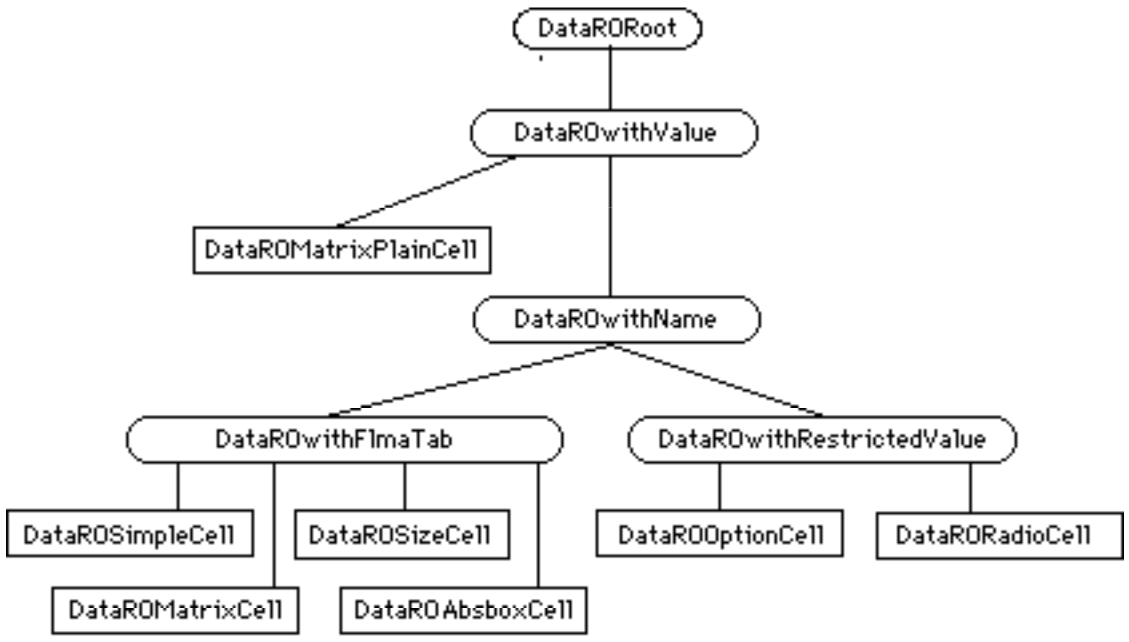


Figure 7: The DataRO hierarchy of referenceable objects

3.5.1 The cached RO model

Five abstract classes form the backbone of the hierarchy, providing increasingly more behavior as one descends the tree. These are:

- DataRORoot
- DataROwithValue
- DataROwithName
- DataROwithFmlaTab
- DataROwithRestrictedValue

The leaves of the tree represent the seven concrete classes from which objects are instantiated. When an RO is created as with a message from Lisp to add or load, it is the DataRO that is instantiated. A form keeps track of its DataROs. Upon creation or changing an existing DataRO, a call to NotifyGui(), just as in the DataForm, causes a GuiRO to come into existence or be repainted. Although there is currently only one view, or type of GuiRO for each DataRO, a list is kept for future decisions about having more than one way to view the data found in the model. This way, a call to NotifyGui() would alert all views to be updated.

The DataRORoot is the abstract class at the root of the DataRO hierarchy. When an RO is created, the DataRORoot is responsible for caching the information from the Lisp model. This could be either default data from an add call, or could be data Lisp read from disk if this RO is part of a newly-loaded form. If the RO was loaded from disk, the left, top, width, and height are cached in the LastSaved data members of the DataRORoot. As with forms, these values are kept by the Java GuiRO until such time as the changed values are flushed to the LastSaved ones before saving the session to disk, or undisplaying a form. The DataRORoot also contains a host of accessor and mutator methods for manipulating the cached data members when requested to do so by the underlying Lisp model.

Each of the other abstract classes add data members and their accompanying accessor/mutator methods relevant for that level of the hierarchy. DataROwithValue contains the formulaString and cached value from Lisp as data members. The value is a Dycon type and will be discussed below. Methods named setValue() are implementations for asking Lisp to evaluate the formula string given by the user. The method named LispSetValue() uses the parsed message from Lisp to actually cache the value of the RO to the returned Dycon. DataROwithName is the abstract class that provides the functionality of a name to an RO. DataROwithFmlaTab is the class that holds the Boolean variable responsible for telling if the formula tab is visible or not. DataROwithRestrictedValue provides the root of the option and radio ROs. Its major function is to hold the option list of strings that the user may choose.

The concrete classes, when created by the NotifyGui() method, contains data members with their associated accessor/mutator methods. The seven classes are:

- DataROMatrixPlainCell
- DataROMatrixCell
- DataROSizeCell
- DataROSimpleCell
- DataROAbsboxCell
- DataROOptionCell
- DataRORadioCell

Although DataROMatrixCell is subclassed from DataROwithFmlaTab, it has unique properties in that it is a container itself of cells. It also contains two DataROSizeCells whose values determine the number and position of these interior cells. The cells within this 'Matrix' are DataROMatrixPlainCells, as seen from figure 7, the only concrete class to extend from DataROwithValue. Indeed, their only functionality is to contain a value.

DataROSimpleCell and DataROAbsboxCell are similar with the exception that the Absbox cell contains a string list of the internal cells cached from data read from disk, as well as a table of the actual DataRO objects added later at the time of their creation. The Option and Radio cells merely provide their NotifyGui() methods the proper instantiation of their respective Gui counterparts.

3.5.2 the RO view

Figure 8 below shows the same structure for the GuiRO hierarchy. The rounded boxes indicate abstract classes while the mitered boxes represent the concrete classes the are instantiated.

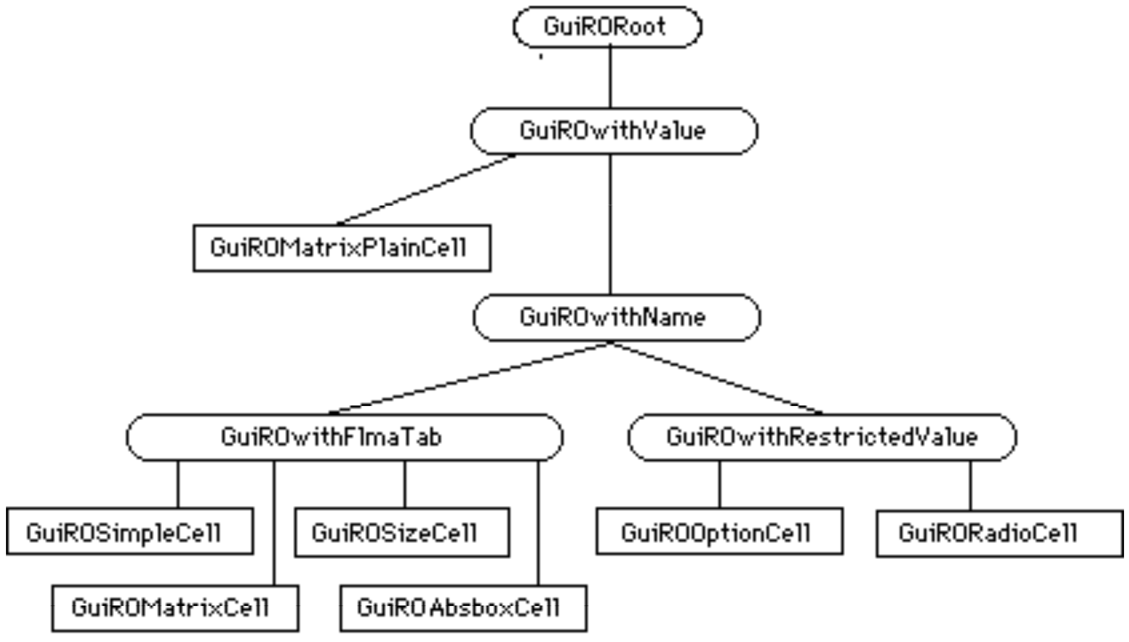


Figure 8: The GuiRO hierarchy of referenceable objects

As with the cached data model, their are five abstract classes forming the backbone of the view hierarchy:

- GuiRORoot
- GuiROwithValue
- GuiROwithName
- GuiROwithFmlaTab
- GuiROwithRestrictedValue

At the root of the RO view is GuiRORoot, which contains the common data important to objects seen in the FormDrawWindow of a form. In fact one of these data is a reference to the FormDrawWindow upon which the object resides. Also, as mentioned above, the left, top, width, and height are found in this abstract class and there is a method for flushing these values to the cached model when appropriate. There are many methods for accessing and changing these values as well as methods that return a Boolean value based on whether a mouse click's coordinates can be found within the object's boundaries.

As one might surmise, GuiROwithValue is mainly responsible for displaying the value of a cell. During an update of the screen a call to this class' method drawValue() is made where the current cached value, a Dycon, is retrieved from the cached model and asked to draw itself at the appropriate coordinates. Because it is at this level of the hierarchy that concrete classes are first instantiated, most of the drawing methods on the view side have been moved up to this class. Since dependency arrows are important to all cells with values, the processing and drawing of arrows are also found in this class. Boolean flags important to arrows and areas within drawn objects can also be found here along with their accessors/mutators.

The name of a cell is maintained as a string and drawn by a method in GuiROwithName. The coordinates of the drawn name are also kept here as part of the information necessary to define a name area for each cell having one. One can query the cell to ask if a mouse click is in a cell's name area by calling this class' inName() method.

Methods responsible for drawing all objects that are part of a formula tab are found in GuiROwithFmlaTab. This includes drawing the tab itself, the drawing and posting of a formula window, and the darkening of the formula window border when a mouse is dragged through it. The drawCellResizer() method is overridden from GuiROwithValue to include the additional area of the name and formula tab when the cell is selected. The Java Font Metrics class is used to ascertain width and height characteristics of the formula string to determine the appropriate width and height to use when drawing the enclosing formula window. All the appropriate Boolean flags and queries for questions relative to formula tab processing are to be found in this class. Four of the seven concrete GuiRO classes are subclassed from GuiROwithFmlaTab and as such a main method for drawing is found here. The method draw() is responsible for testing the condition of all the relevant Boolean flags and based on their value, call the drawing methods previously discussed in this and other classes higher up in the hierarchy to do the actual drawing.

GuiROwithRestrictedValue is at the root of both the option and radio cell Gui objects. There is a fundamental difference in how the value of one of these cells is displayed. Instead of the value being drawn in the FormDrawWindow as with cells, the value resides in a Java object, the Checkbox. Being an actual Java Abstract Window Toolkit component, it is not drawn but rather placed in the FormDrawWindow container. Any subsequent moving of the cell involves removing and replacing the component at new coordinates along with any ancillary drawing of other attributes.

The seven concrete classes of the GuiRO are all instantiated via a call to NotifyGui() made the corresponding DataRO. The parameters of the constructor provide a reference to the FormDrawWindow, a reference to the underlying DataRO, and the left and top coordinates of the drawn object. If the lastSaved values of the width and height are zero, indicating the cell is being added and not read from disk, default values are given, otherwise values read from disk are used. These concrete classes are:

- GuiROMatrixPlainCell
- GuiROMatrixCell
- GuiROSizeCell
- GuiROSimpleCell
- GuiROAbsboxCell
- GuiROOptionCell

- GuiRORadioCell

Although much of the drawing is common to all cell types, other than GuiROSimpleCell and GuiROSizeCell, each has unique qualities that require overriding or augmenting existing drawing methods. GuiROMatrixPlainCell, being a subclass of GuiROwithValue, has simply fewer Boolean flags to query and less to draw.

GuiROAbsboxCell must deal with values much differently and overrides drawValue() to handle the case where the cell is hidden; requiring the display of the simple error if there are no cells in the absbox at the moment, or the imageDycon is cells are present. If the Absbox is not hidden, it draws the interior cells. An additional method exists to draw those interior cells.

GuiROOptionCells and GuiRORadioCells have a host of methods used for creation of, adding to, removing from, moving around, setting visibility of, and selecting the value of the Java checkbox component. The radio buttons are just a form of mutually exclusive checkboxes as part of a checkbox group, and the methods are similar to the checkbox of the option cell. Both contain the same list of options from which a value is selected. The size of a GuiRORadioCell is determined by the number of buttons currently displayed.

Because a GuiROMatrixCell is a collection of several other objects, it is the most expanded concrete class. In its draw method, it first calls the draw method of the superclass, then based on Boolean values, it may draw the two size cells, the top and side of the region overlay, any matrix regions that may be present, and any interior GuiROMatrixPlainCells that it may contain. If the matrix itself is moved, methods are overridden in this class to recalculate the position of any of the associated objects at the same time. Mutating the "selected" Boolean flag is overridden to select the contained size cells as well. The evaluation of a matrix results in a list of associated pairs of cellIDs and their current value being returned from the Lisp evaluation engine. Consequently, the drawValue() method is overridden to iterate over this list, update the value of each internal cell, and ask it to draw itself. As mentioned, a matrix contains what are called matrix regions. A MatrixRegion class provides the matrix with a unique way to give its internal cells a formula.

This special class, not strictly part of the GuiRO hierarchy, will be discussed in the next section of topics.

3.6 Other Objects

The following subsections address several topics that are integral to understanding the full implementation of Forms/3 in Java, but are themselves loosely coupled. They do not form part of an ancestral hierarchy, yet interact with members of the hierarchies discussed above.

3.6.1 Matrix Regions

Class MatrixRegion contains methods that provide a mechanism that allows the user to give formulas to cells within a matrix. It is responsible for maintaining the ending coordinate of a matrix region as well as the formula string and general formula for cells within it. Matrices in Forms/3 are containers of cells, called MatrixPlainCells. All cells within a matrix may have the same formula if they are part of the same matrix region. Elements of the MatrixRegion class are similar to class GuiROwithFmlaTab in that a matrix region also has a formula tab and methods for drawing the tab and other region related objects such as the vertical region creation line. When a formula is provided by the user, it is sent to the Lisp side for evaluation. Since a region's formula applies to multiple cells, the return message from Lisp is a list of pairs of internal cellIDs and their respective values (Dycons, see below). This returned list is not seen by the matrix region nor the internal cells themselves but rather by the matrix, which updates the value of each internal cell and asks it to draw itself. This is an example of where the asynchrony of message passing is illustrated in part, wherein the destination of the returned message is not the same object that sent the request. The Lisp engine is responsible for reevaluating the various regions of a matrix. This includes both the one currently being evaluated and the existing regions, producing the evaluated list of pairs. There is a method in this class for parsing information sent from lisp about existing regions (such as if the matrix was read from disk). The user can create a new matrix region by dragging out the region creation bar and upon release of the mouse, the ending coordinates of the region are determined.

3.6.2 Dycons

The word Dycon is an abbreviation for "Dynamic Icon". It is a result of the evaluated formula returned from the Lisp model; a value. Each dycon class is responsible for encapsulating the methods needed for drawing the type of value it contains. Class Dycon is an abstract class from which all the other Dycons are subclassed. There are two abstract methods that each subclass must implement: getStringValue() and draw(). The Dycon class itself contains a complete list of attributes any one subclassed Dycon may have and implements a parsing method to assign these attributes to the Dycon. All the subclassed Dycons, except for the MatrixDycon, which has no analog, can be found in the Lisp model. The superclass and its subclasses are listed below:

- Dycon
- AbsDycon
- BooleanDycon
- BoxDycon
- CircleDycon
- ComposeDycon
- ErrorDycon
- LineDycon
- MatrixDycon
- NoValueDycon
- NumberDycon
- TextDycon
- UnknownDycon

As seen in the list, there are Dycons for text, numbers, and Boolean values as one might expect. There are also Dycons for graphical types such as circles, boxes, and lines. An ErrorDycon is an error value that may result if the user should enter a formula that generates an error. A DataRO may exist but has not yet been given a formula, in which case an evaluation of the cell will produce the NoValueDycon. An UnknownDycon displays a question mark if evaluation of the cell has not yet been performed. Three Dycons, the AbsDycon, ComposeDycon and MatrixDycon require additional explanation and will be covered in the following paragraphs.

Something new was required for Java to obtain the information needed to display the values of a matrix's cells, a MatrixDycon. As mentioned, there is no MatrixDycon on the Lisp side because matrices do not have values, they are simply containers of cells. At the time of writing this paper all maintenance, processing, and displaying of matrix internal cells, matrix regions, and the matrix itself is tightly interwoven between the Garnet user interface and the Lisp model. A special dycon temporarily implemented for other reasons was found to contain the content and format of a matrix region's evaluated formula. The machinery for processing such information on the Java side already existed in the pattern of Dycons. It seemed only logical that this information be packaged into a Dycon type and used as the value of a matrix (i.e. a list of cellIDs and their associated Dycons). It remained only for the draw method of class MatrixDycon to iterate over this list and ask each internal cell to update (draw) itself.

AbsDycons are instances of user defined types (defined by users via Vadt forms). They maintain a parts list of internal cellIDs paired with their Dycon much like the MatrixDycon above. This paired list is implemented as a hashtable, keyed on the cellID. The appearance of the AbsDycon is determined by a data member holding the value of a Dycon. The name of this data member follows the underlying Lisp convention of ImageDycon although the name does not reflect a new type of Dycon class. Instead, it reflects a Dycon that is the value of the Image cell on the VadtForm that created this AbsDycon.

A ComposeDycon contains a list of dycons that are composed together into a "group picture" (such as a box centered inside a circle with a text label to the right). This time each Dycon is paired with a set of left and top coordinates. The draw method of a ComposeDycon is the most elaborate. Here, each Dycon is asked to draw at its own set of coordinates relative the upper left corner of the cell containing the ComposeDycon. Most often, although not always, the type of an ImageDycon data member in an AbsDycon is a ComposeDycon.

3.6.3 Arrows

Objects of class Arrow are drawn from one RO to another. A single mouse click by the user on the middle mouse button while the cursor is positioned over an RO generates a message to Lisp asking for a list of ROs whose formula the current RO may affect, if any. This "affecting" relationship is known as a dependency in Forms/3 and may be shown visually as a dependency arrow.

The list of ROs in the returned message, is iterated over and an Arrow is instantiated by passing the source RO and destination RO as parameters to the constructor of the Arrow object. For flexibility an additional constructor is provided that takes as parameters, the x and y coordinates of the beginning and ending of an arrow position. If ROs are given, the Arrow object itself determines the coordinates of the arrow, including the situation where and arrow may span more than one form. The object also finds the middle of the arrow for drawing a "remove" box. Draw methods are found in this class for displaying the arrow on one or more forms, drawing the arrowhead, and for removing them as well.

3.6.4 The Global class

Since, in Java, there is no notion of "global-ness", everything must be wrapped in a class. The class Global is designed to provide a repository of constants, variables, data structures, and methods that may be accessed by any other object. Public constants and variables in this class begin with a "$" character. Two data structures are provided, one called the FormTable holds the actual form objects and the other, the FormsList, provides a list of strings representing the currently loaded forms. This list is displayed in a separate window and can be hidden by the user. Methods exist to provide easy access to any form currently loaded. This becomes particularly useful for the MessageHandler in finding the object for which a message is directed. Two methods allow any object to request that all ROs on a form or all ROs on all forms be evaluated. A very useful method found in this class is called getGuiCell() which when given a cellID as a parameter, will return the actual RO. There are several methods that return the Boolean value "true" when the given cellID represents an RO of the type for which the method is testing. The method makeDycon() will instantiate and return a Dycon based on the tokenized string given to it. Finally, there are methods that parse the formula string and general formula from a given tokenized string. Since the need for this functionality is required in different situations (such as parsing directly from disk, posting a new formula provided by the user, or creating a new matrix region), it seemed natural to localize the code in one place.

3.6.5 The ROFactory

Whenever a file is read from disk and sent to the Java implementation, any number of the seven types of GuiRO objects may need to be created. It seemed logical to localize this creation in one place. Class ROFactory is responsible for providing methods to handle the instantiation of six of the seven GuiROs (since information about the number of MatrixPlainCells a Matrix may have is not found in a disk file, rather secondarily from information gleaned from the SizeCells, their instantiation is left to the Matrix). Much of the data is common to all types of cells and factored out to separate methods. The constructor of an ROFactory is given a reference of the form for which these cells are being made and after instantiation, are appended to the form's DataROTable.

3.6.6 Dialogs

A large part of a user's interaction with an application that provides a graphical user interface is with dialog boxes. This implementation has several that deal with specific types of interaction. The following is a list of the dialogs currently found in this implementation:

- CellAttributes
- EditOptionsDialog
- FormulaDialog
- InfoDialog
- NameDialog
- QuestionDialog

All Dialog classes are subclassed from the Java class Dialog. InfoDialog is the simplest, merely giving the user some bit of information and requiring no response save for its undisplay. The QuestionDialog waits for a yes/no answer to some posted question usually setting a Boolean variable as a response. The NameDialog allows the user to type in a name for an RO. Each RO has several attributes that the user may choose to display or hide, such as the border, name, the formula tab if any, or even the cell itself. A

CellAttributes dialog box is customized for each RO type and allows the user to set Booleans related to that ROs attributes. The EditOptionsDialog allows for the user to change the list of options of an OptionCell or RadioCell anytime during the user session.

The Java class Dialog is in turn subclassed from the Java class Frame. This allows the dialog box to have a modality. Most of the classes have their modality set to true preventing the user from continuing until the dialog has been closed. FormulaDialog has its modality set to false to allow the user to click on a cell and provide a reference to a formula he or she may be editing at the time, eliminating the need for a textual input of that reference.

3.7 Java Containers, Inheritance, and Casting

Java is a language wherein all objects have a common root, the Java Class Object. In this application as well as others, there is a need to place objects into a container and pull them back out at a later time. These containers, such as hash tables or vectors allow for growth at runtime and is advantageous because one does not know at runtime how many objects there will be. Anything can be put into the container because at the time of inclusion the object is upcast to the common root, Object. Objects of all levels in the RO hierarchy are placed into containers. A problem occurs when one pulls the object out of the container. It has essentially lost its identity. Every time the programmer pulls out an object, he or she must downcast it to the appropriate class [Eckel 1998]. This means that the programmer must know ahead of time what class the object might be. At runtime that knowledge is not always available and a test of the object's identity is required. This test is performed with the Java keyword "instanceof". A programmer will discover many times in the code where this testing appears and make note that it is not a design decision on the part of the programmer but a feature of the Java language.

4. Summary

Forms/3, a visual programming language implemented in Lisp, currently uses a graphical user interface written in Garnet. Garnet is a constraint-based language itself written in Lisp. Garnet seemed a good choice for the job of graphical representation but unfortunately, it is no longer supported. This paper discusses a graphical interface written in Java for Forms/3. In addition to covering the actual implementation of the interface, this paper also discusses the major design decisions involving the Model/View/Controller architecture, the asynchronous socket communication required, and a few design patterns recognized as important in the structure of the code.

5. Future Work

The Java implementation of the graphical interface is far from being done. This paper represents a beginning that hopefully will be continued by many others to come. There are several items that would be important in the near future for the purpose of running the standard tutorial programs provided by the Lisp implementation. Including the time dimension, animation, and matrices would allow users to take the Forms/3 tutorial. Although the code for implementing matrices is written on the Java side, at the time of this writing the Lisp engine was not sufficiently untangled from the Garnet interface to allow testing of the code. This situation should be remedied soon.

In the long term, the entire graphical user interface should be switched over to Java, after which the underlying Lisp model could then be converted as well.

6. Appendices

Appendix A

JavaDocs

The following is a text saved copy of the html web page containing the class hierarchy of the project. It is included here for completeness. A better visual appearance can be seen at these URLs:

- http://www.cs.orst.edu/~hackenda/javaforms/docs/tree.html
- file:/nfs/spectre/u2/burnett/Resrch/Forms/web/javaforms/docs /tree.html

Clicking on any of the class Names will send the user to a Java Document page in the same format as that seen with the Java API at Sun Microsystems.

```
All Packages  Index

------------------------------------------------------------------------

Class  Hierarchy

  * class java.lang.Object
          o class Arrow
          o class java.awt.Component (implements java.awt.image.ImageObserver,
            java.awt.MenuContainer, java.io.Serializable)
              + class java.awt.Container
                  + class gjt.DoubleBufferedContainer
                      + class FormDrawWindow (implements
                        java.awt.event.MouseListener,
                        java.awt.event.MouseMotionListener)
                  + class java.awt.Panel
                      + class BorderedPanel
                  + class java.awt.Window
                      + class java.awt.Dialog
                          + class CellAttributes (implements
                            java.awt.event.ActionListener)
                          + class EditOptionsDialog (implements
                            java.awt.event.ActionListener)
                          + class FormulaDialog (implements
                            java.awt.event.ActionListener)
                          + class InfoDialog (implements
                            java.awt.event.ActionListener)
                          + class NameDialog (implements
                            java.awt.event.ActionListener)
                          + class QuestionDialog (implements
                            java.awt.event.ActionListener)
                      + class java.awt.Frame (implements
                        java.awt.MenuContainer)
                          + class Forms/3 (implements
                            java.awt.event.ActionListener)
                          + class FormsListGui
                          + class GuiSimpleForm (implements
                            java.awt.event.ActionListener,
                            java.awt.event.ComponentListener)
                              + class GuiVadtForm
                          + class HelpWindow (implements
                            java.awt.event.ActionListener)
                          + class SplashScreen
              + class DataRORoot
                  + class DataROwithValue
                      + class DataROMatrixPlainCell
                      + class DataROwithName
                          + class DataROwithFmlaTab
                              + class DataROAbsboxCell
                              + class DataROMatrixCell
                              + class DataROSimpleCell
                              + class DataROSizeCell
                          + class DataROwithRestrictedValue
                              + class DataROOptionCell
                              + class DataRORadioCell
          o class DataSimpleForm
              + class DataVadtForm
          o class Dycon
              + class AbsDycon
              + class BooleanDycon
              + class BoxDycon
              + class CircleDycon
              + class ComposeDycon
              + class ErrorDycon
              + class LineDycon
              + class MatrixDycon
              + class NoValueDycon
              + class NumberDycon
              + class TextDycon
              + class UnknownDycon
          o class FormulaStringHandler
          o class Global
          o class GuiRORoot
              + class GuiROwithValue
                  + class GuiROMatrixPlainCell
                  + class GuiROwithName
                      + class GuiROwithFmlaTab
                          + class GuiROAbsboxCell
                          + class GuiROMatrixCell
                          + class GuiROSimpleCell
                          + class GuiROSizeCell
                      + class GuiROwithRestrictedValue
```

```
                    + class GuiROOptionCell (implements
                       java.awt.event.ItemListener)
                    + class GuiRORadioCell (implements
                      java.awt.event.ItemListener)
o class LispParser
o class LispTalker
o class MatrixRegion
o class java.awt.MenuComponent (implements java.io.Serializable)
     + class java.awt.MenuItem
          + class java.awt.Menu (implements java.awt.MenuContainer)
               + class RadioMenu (implements
                 java.awt.event.ItemListener)
o class MessageHandler
o class ROFactory
o class java.lang.Thread (implements java.lang.Runnable)
     + class LispListener
o class java.util.Vector (implements java.lang.Cloneable,
  java.io.Serializable)
     + class DataROTable
```

Appendix B

<u>**The Code**</u>

At these URLs:

- http://www.cs.orst.edu/~hackenda/javaforms/code/code.html
- file:/nfs/spectre/u2/burnett/Resrch/Forms/web/javaforms/code /code.html

one can find the following list of the 65 java source files. If one clicks on any of these filenames, he or she is sent to a page containing the source code in read only mode. The following is a text saved copy of that page followed by three examples of source code to illustrate the style and coding conventions used by the author.

----

[o] AbsDycon.java
[o] Arrow.java
[o] BooleanDycon.java
[o] BorderedPanel.java
[o] BoxDycon.java
[o] CellAttributes.java
[o] CircleDycon.java
[o] ComposeDycon.java
[o] DataROAbsboxCell.java
[o] DataROMatrixCell.java
[o] DataROMatrixPlainCell.java
[o] DataROOptionCell.java
[o] DataRORadioCell.java
[o] DataRORoot.java
[o] DataROSimpleCell.java
[o] DataROSizeCell.java
[o] DataROTable.java
[o] DataROwithFmlaTab.java
[o] DataROwithName.java
[o] DataROwithRestrictedValue.java
[o] DataROwithValue.java
[o] DataSimpleForm.java
[o] DataVadtForm.java
[o] Dycon.java
[o] EditOptionsDialog.java
[o] ErrorDycon.java
[o] FormDrawWindow.java
[o] Forms/3.java
[o] FormsListGui.java
[o] FormulaDialog.java
[o] FormulaStringHandler.java
[o] Global.java

[o] GuiROAbsboxCell.java
[o] GuiROMatrixCell.java
[o] GuiROMatrixPlainCell.java
[o] GuiROOptionCell.java
[o] GuiRORadioCell.java
[o] GuiRORoot.java
[o] GuiROSimpleCell.java
[o] GuiSizeCell.java
[o] GuiROwithFmlaTab.java
[o] GuiROwithName.java
[o] GuiROwithRestrictedValue.java
[o] GuiROwithValue.java
[o] GuiSimpleForm.java
[o] GuiVadtForm.java
[o] HelpWindow.java
[o] InfoDialog.java
[o] LineDycon.java
[o] LispListener.java
[o] LispParser.java
[o] LispTalker.java
[o] MatrixDycon.java
[o] MatrixRegion.java
[o] MessageHandler.java
[o] NameDialog.java
[o] NoValueDycon.java
[o] NumberDycon.java
[o] QuestionDialog.java
[o] ROFactory.java
[o] RadioMenu.java
[o] SplashScreen.java
[o] TextDycon.java
[o] UnknownDycon.java

Example 1: A DataRO class - DataROMatrixPlainCell

```
import gjt.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
*This is the only concrete class that currently extends DataROwithValue. It is
*the cell inside a matrix and only has a value.
*/
public class DataROMatrixPlainCell extends DataROwithValue {

 private TextField        textfield = new TextField("");
 protected int            rowNum;
 protected int            colNum;
 /**
 *The x and y position and formdrawwindow are passed on to the parent class
 *DataRORoot.
 */
```

```
    public DataROMatrixPlainCell(int x, int y) {
        super(x, y);
    }
    /**
    *Given: nothing
    *Creates a Gui the first time it is used, otherwise it checks a list of
    *Guis' and asks them to update(draw) themselves.
    */
    public void NotifyGui(FormDrawWindow fdw, Graphics g) {
        if (guiList.size() == 0) {
            System.out.println("creating new GuiRO...");
            myGui = new GuiROMatrixPlainCell(fdw, this, getLastSavedLeft(),
                                                        getLastSavedTop());
            guiList.addElement(myGui);
            myGui.upDate(g);
        }
        else {
            for(int i=0; i < guiList.size(); i++) {
                GuiROMatrixPlainCell tempcell =
(GuiROMatrixPlainCell)guiList.elementAt(i);
                tempcell.upDate(g);
            }
        }
    }
    //accessors, mutators
    public void setRowNum(int i) {
        rowNum = i;
    }
    public int getRowNum() {
        return rowNum;
    }
    public void setColNum(int i) {
        colNum = i;
    }
    public int getColNum() {
        return colNum;
    }
  }//end  DataROMatrixPlainCell
  --------------------------------------------------------------------------
```

Example 2: The GuiRO for the DataRO above - GuiROMatrixPlainCell

```
  import gjt.*;
  import java.awt.*;
  import java.awt.event.*;

  /**
  *This is the only concrete class that currently extends GuiROwithValue.
  */
  public class GuiROMatrixPlainCell extends GuiROwithValue {
    /**
    *Given: a formdrawwindow, the RO that created it, and the position
    *in the window; sets the RO to myData.
    */
```

```
public GuiROMatrixPlainCell(FormDrawWindow c, DataROMatrixPlainCell cell,
                                   int left, int top) {
    super(c, left, top);
    myData = cell;
    if (myData.getLastSavedWidth() == 0) {
        setInitialWidth(40);
    }
    else {
        setInitialWidth(myData.getLastSavedWidth());
    }
    if (myData.getLastSavedHeight() == 0) {
        setInitialHeight(40);
    }
    else {
        setInitialHeight(myData.getLastSavedHeight());
    }
}
/**
*Given: nothing. Cast the Data object to the appropriate level of the
*hierarchy. Returns that object as a DataROMatrixPlainCell .
*/
private DataROMatrixPlainCell   MyData() {
    return (DataROMatrixPlainCell) myData;
}
/**
*Given: the Grphics object to draw into.
*draws the cell according to the boolean values associated with the cell
*attributes, as well as arrows and the value of the cell.
*/
public void draw(Graphics g) {
    DataROMatrixPlainCell tmpData = MyData(); //cast the level
    if (tmpData.isCellHidden() &&
getDrawWindow().getMyForm().myGuiForm.hideAll) {
        return;
    }
    else {
        if (!tmpData.isBorderHidden()) {
            drawCell(g);
        }
        if (isArrowOn()) {
            //first grab the Graphics object passed in from the mouse click
            arrowG = g;
            //now send a request for a list of cells to draw to if first time
            if (requestFlag) {
                requestArrowList();
                requestFlag = false;
            }
            else {//just draw them if after first time
                drawArrows();
            }
        }
        if (isArrowFromOn()) {
            drawArrowFrom(g);
        }
```

```
            drawValue(g);
        }
  }//end draw
   /**
   *Given: a mouse click position.
   *if used, a boolean but this method is a dummy and never used by this class
   */
   public boolean inName(int x, int y) {return false;}
}//end GuiROMatrixPlainCell
--------------------------------------------------------------------------
```

Example 3: The main Forms/3 class containing code to start up Lisp and create the main Form - Forms/3

```
import gjt.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.net.*;

/**
 *This is the top level class that contains the public static void
 *main(String args[]) method. Typing "java Forms/3" at the command line
 *initiates the splashscreen and main form and displays them sequentially.
 *The actionlistener is the interface by which the mouse events are monitored
 *and appropriate action taken.
 */
public class Forms/3 extends Frame implements ActionListener {

  private String              dirName;
  private InfoDialog          initInfo;
  private Dimension           scrnSize;
  private Frame               myFrame;
  private StringTokenizer     st;
  private FormsListGui        formsListGui;
  private MessageHandler      msgHandler;
  private static Toolkit      toolkit;
  private static int          port;
  private static String       java;
  private static String       username;
  private String              lastDirectory = "";

  /**
  *This is the "main" method which initiates everything.
  *It shows a splashscreen and creates the main form and displays it
  *at the location calculated in the setLocation method of the frame.
  */
  public static void main(String args[]) {

      if (args.length > 0) {
          Integer i = new Integer(args[0]);
          port = i.intValue();
          java = args[1];
```

```
            username = args[2];

        }
        SplashScreen ss = new SplashScreen();
        ss.setVisible(true);
        ss.dispose();

        Forms/3 forms3 = new Forms/3("Forms/3");
        forms3.setSize(200, 154);
        Dimension   scrnSize  = toolkit.getScreenSize();
        forms3.setLocation(scrnSize.width/4  - 100,
                                    scrnSize.height/4 - 200);
        forms3.setVisible(true);
    }//end "main"

    /**
    *Given: a string to name the frame.
    *It instantiates the menu, slider, buttons, and initializes a vector
    *to hold the form objects and a list to display their names if loaded.
    *It also instantiates the lispTalker, lispListener, and Messagehandler
    *classes to support the socket activities required to interact with the
    *underlying lisp engine.
    */
    public Forms/3 (String s) {

        super(s);

        MenuBar mbar                = new MenuBar();
        Menu fileMenu               = new Menu("File");
        Menu editMenu               = new Menu("Edit");
        Menu viewMenu               = new Menu("View");
        Menu helpMenu               = new Menu("Help");
        Menu newFormMenu            = new Menu("New Form...");
        myFrame                     =  Util.getFrame(this);
        toolkit                     =  Toolkit.getDefaultToolkit();

        //this attempts to freeze the frame size
        myFrame.setResizable(false);

        //create the Global FormTable
        Global.FormTableInitialize();

        //create the Global FormsList passing a reference for frame usage
        Global.ListInitialize(this);

        //now create the Gui to hold the currently loaded forms list
        formsListGui            = new FormsListGui();

        //create time slider
      Slider timeSlider = new Slider(1, 10, 1, 300);

        //create buttons and buttonpanel
        Button button1          = new Button("start");
        Button button2          = new Button("stop");
```

```
Button button3              = new Button("reverse");
ButtonPanel bp              = new ButtonPanel();

//these buttons will listen to mouse events
button1.addActionListener(this);
button2.addActionListener(this);
button3.addActionListener(this);

//create menus
MenuItem simpleFormItem        = new MenuItem("Simple Form");
MenuItem vadtFormItem          = new MenuItem("VADT Form");
MenuItem templateFormItem      = new MenuItem("Template Form");
newFormMenu.add(simpleFormItem);
newFormMenu.add(vadtFormItem);
newFormMenu.add(templateFormItem);
fileMenu.add(newFormMenu);

MenuItem loadItem           = new MenuItem("Load Form");
MenuItem unloadItem         = new MenuItem("Unload Form");
MenuItem saveItem           = new MenuItem("Save Form");
MenuItem displayItem        = new MenuItem("Display Form");
MenuItem undisplayItem      = new MenuItem("Undisplay Form");
MenuItem quitItem           = new MenuItem("Quit");
MenuItem helpItem           = new MenuItem("General Help");
MenuItem copyItem           = new MenuItem("Copy Form");

fileMenu.add(loadItem);
fileMenu.add(unloadItem);
fileMenu.add(saveItem);
fileMenu.add(displayItem);
fileMenu.add(undisplayItem);
fileMenu.addSeparator();
fileMenu.add(quitItem);

helpMenu.add(helpItem);

editMenu.add(copyItem);

MenuItem viewItem           = new MenuItem("View Forms List");
MenuItem hideItem           = new MenuItem("Hide Forms List");
viewMenu.add(viewItem);
viewMenu.add(hideItem);

//these items listen to mouse events
loadItem.addActionListener(this);
unloadItem.addActionListener(this);
saveItem.addActionListener(this);
viewItem.addActionListener(this);
hideItem.addActionListener(this);
displayItem.addActionListener(this);
undisplayItem.addActionListener(this);
simpleFormItem.addActionListener(this);
vadtFormItem.addActionListener(this);
templateFormItem.addActionListener(this);
```

```
        quitItem.addActionListener(this);
        helpItem.addActionListener(this);
        copyItem.addActionListener(this);

        //add menus to menubar
        mbar.add(fileMenu);
        mbar.add(editMenu);
        mbar.add(viewMenu);
        mbar.add(helpMenu);
        mbar.setHelpMenu(helpMenu);
        setMenuBar(mbar);

        //arrange all items in the frame using borderlayout
        setLayout(new BorderLayout());
        add(timeSlider,  "South");
        bp.add(button1);
        bp.add(button2);
        bp.add(button3);
        add(bp,  "North");

        //put up an info window to tell user about loading lisp files
        informUser();

        //create client socket and get lisp side going
        LispTalker.Start(port);
        LispTalker.send("(USERNAME \"" + username + "\")");
        LispTalker.send("(load  \"RUN\")");
        LispTalker.send("(run :JAVA " + java + ")");
        LispTalker.send("(load  \"gui.lisp\")");
        //get the listener going
        LispListener.Start();
        //just so we can pass a reference of Forms/3 to it
        MessageHandler.Start(this);

        //make the frame closable
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                dispose();
                System.exit(0);
            }
        });
    }//end constructor
    /**
    *Given: nothing. Simply puts up an info dialog box to inform the user that we
    *need to wait until the lisp files are loaded. Disappears when Gui.lisp is
    *done loading.
    */
    private void informUser() {
        initInfo = new InfoDialog(myFrame,
            "Please wait. Loading lisp files...", false);
        initInfo.show();
    }
    /**
    *Given: nothing. Disposes of the dialog box put up in informUser()
```

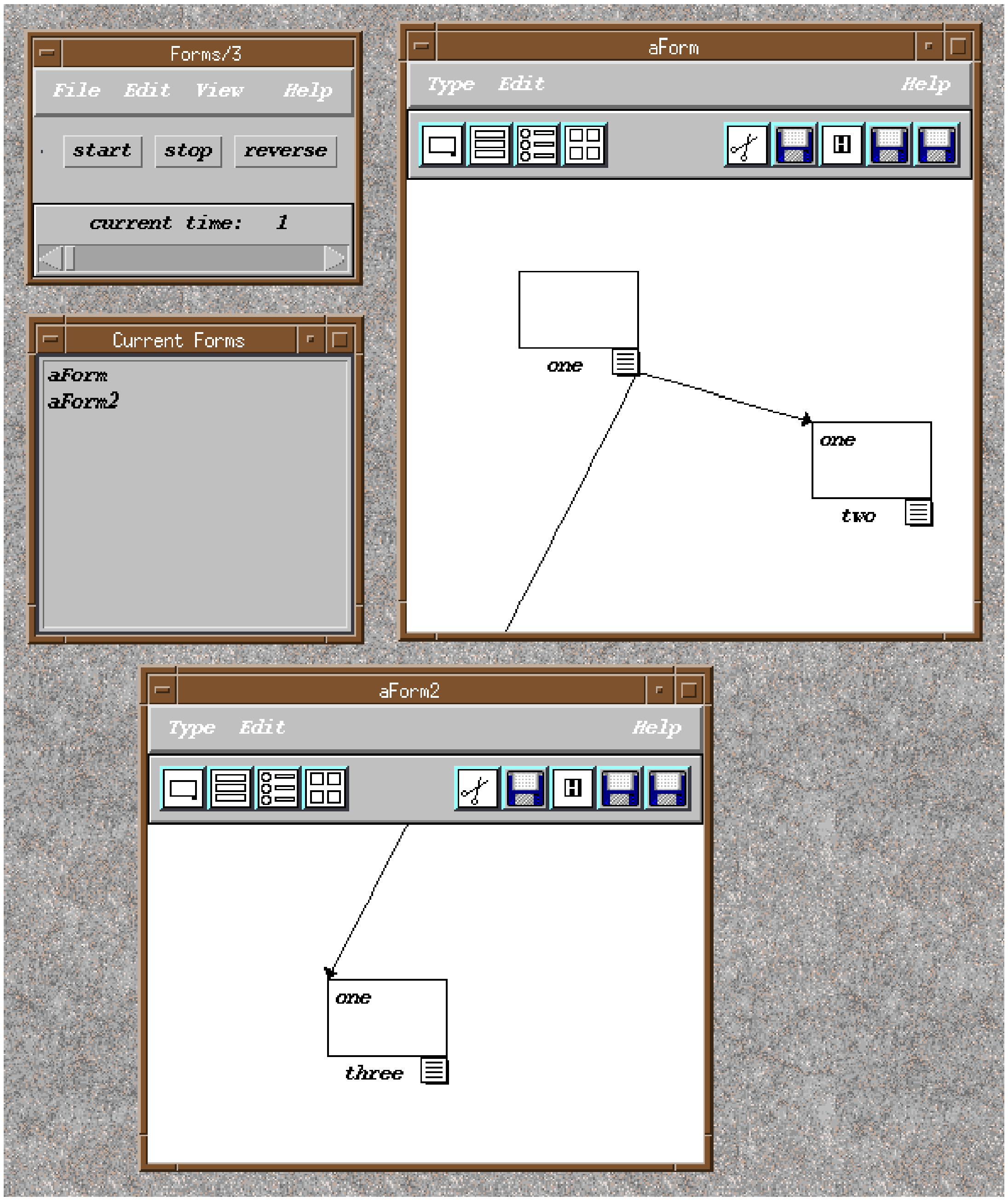
```
*/
public void killInitInfo() {
    initInfo.dispose();
}
/**
*Given: an integer indicating which type of form to create.
*Creates the form by asking lisp to do it with a call to
*gui-new-form.
*/
private void createForm(int i) {
    String s = null;
    Frame myFrame = Util.getFrame(this);
    //UserInput extends a java dialog object
    NameDialog nd = new NameDialog(myFrame,
                    "Form Name", "Give the Form a name: ");
    nd.setTextField("aForm");
    nd.show();
    if (nd.isAccepted()) {
        String formID = nd.getTextField();
        if (Global.checkFormsList(formID)) {
            if (i == 1) {
                //ask lisp to create the Simple form
                LispTalker.send("(gui-new-form \"" + formID + "\" 0)");
            }
            else if (i == 2) {//do nothing yet
                //ask lisp to create the Vadt form
                LispTalker.send("(gui-new-form \"" + formID + "\" 1)");
            }
            else if (i == 3) {//do nothing yet
                //ask lisp to create the Template form
            }
        }
    }
}
/**
*Given: mouse events
*Delegates action from interaction of menu items to appropriate objects
*/
public void actionPerformed(ActionEvent event) {
    String arg    = event.getActionCommand();
    Frame dialogFrame = new Frame();
    //first - the three buttons
    if(arg.equals("start"))  {
        System.out.println("start  works!");
    }
    if(arg.equals("stop"))  {
        System.out.println("stop  works!");
    }
    if(arg.equals("reverse"))  {
        System.out.println("reverse  works!");
    }
    //then the menu options
    if(arg.equals("Quit"))  {
        dispose();
```

```
            LispTalker.send("(quit)");
            System.exit(0);
        }
        if (arg.equals("General Help")) {
            String frameTitle = "General";
            LispTalker.send("(gui-help-file \"" + frameTitle + "\")");
        }
        if(arg.equals("Load Form")) {
            FileDialog dialog =
                new FileDialog(dialogFrame, "Load a File");
            System.out.println("dir: " + dialog.getDirectory());
            if (lastDirectory.equals("")) {
                dialog.setDirectory("/nfs/spectre/u4/hackenda/Forms/");
            }
            else {
                dialog.setDirectory(lastDirectory);
            }
            dialog.show();
            String formID = dialog.getFile();
            dirName  = dialog.getDirectory();
            lastDirectory = dirName;
            if (!formID.equals("")) {
                LispTalker.send("(gui-load-form \"" + dirName + formID + "\" )");
            }
            else {//do nothing
            }
        }
        if(arg.equals("Unload Form")) {
            if (Global.isListEmpty()) {
                return;
            }
            formsListGui.showList();
            String formID = Global.getCurrentListItem();
            if (formID == null) {
                InfoDialog m = new InfoDialog(myFrame, "Please select a Form to unload.",
                true);
                m.show();
                if (m.isOK()) {}
            }
            else if (formID != null) {
                QuestionDialog m = new QuestionDialog(myFrame, "Save changes to "+
formID
                + " before unloading?", true);
                m.show();
                if (m.isOK() && !m.isAbort()) {
                    saveForm(formID);
                }
                else if (m.isAbort()) {
                    return;
                }
                LispTalker.send("(gui-unload-form \"" + formID + "\" )");
            }
        }
        if(arg.equals("Display Form")) {
```

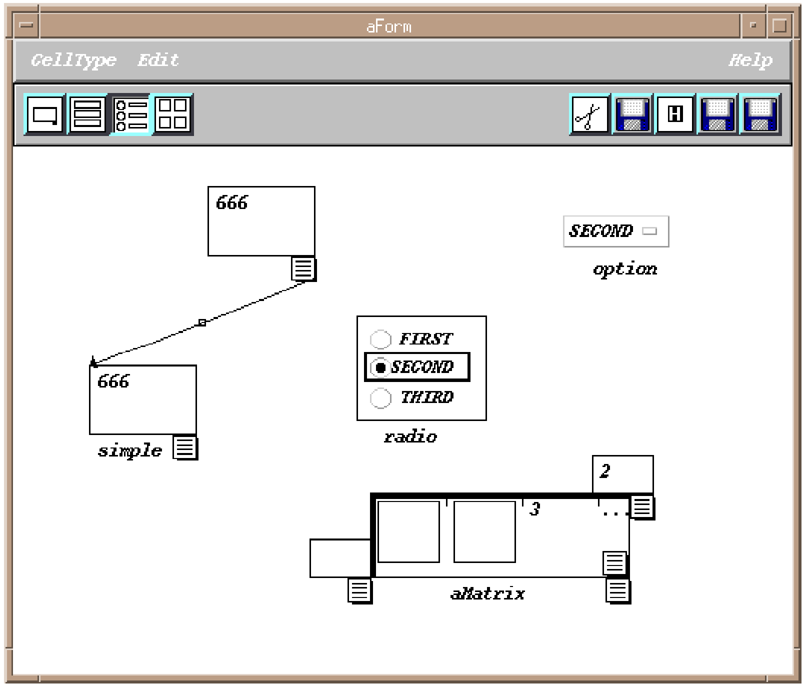
```
        if (Global.isListEmpty()) {
            return;
        }
        formsListGui.showList();
        String s = Global.getCurrentListItem();
        if (s == null) {
            InfoDialog m = new InfoDialog(myFrame, "Please select a Form to display.",
            true);
            m.show();
            if (m.isOK()) {}
        }
        else {
            for (int i=0; i < Global.FormTableGetSize(); i++) {
                String tempFormID = Global.FormTableGet(i).getFormID();
                if (tempFormID.equals(s)) {
                    System.out.println("in Display, notifying Gui...");
                    Global.FormTableGet(i).NotifyGui();
                    LispTalker.send("(gui-display-form \"" +
                                                        tempFormID + "\")");
                }
            }
        }
    }
    if(arg.equals("Undisplay Form")) {
        if (Global.isListEmpty()) {
            return;
        }
        formsListGui.showList();
        String s = Global.getCurrentListItem();
        if (s == null) {
            InfoDialog m = new InfoDialog(myFrame, "Please select a Form to undisplay.",
            true);
            m.show();
            if (m.isOK()) {}
        }
        else {
            for (int i=0; i < Global.FormTableGetSize(); i++) {
                String tempFormID = Global.FormTableGet(i).getFormID();
                if (tempFormID.equals(s)) {
                    System.out.println("in UnDisplay, removing Gui...");
                    Global.FormTableGet(i).undisplayGui();
                    LispTalker.send("(gui-undisplay-form \"" +
                                                        tempFormID + "\")");
                }
            }
        }
    }
    if(arg.equals("View Forms List")) {
        formsListGui.showList();
    }
    if(arg.equals("Hide Forms List")) {
        formsListGui.hideList();
    }
    if (arg.equals("Simple Form")) {
            createForm(1);
        }
        else if (arg.equals("VADT Form")) {
            createForm(2);
        }
        else if(arg.equals("Template Form")) {
            createForm(3);
        }
        if(arg.equals("Save Form")) {
            if (Global.isListEmpty()) {
                return;
            }
            formsListGui.showList();
            String formID = Global.getCurrentListItem();
            if (formID == null) {
                InfoDialog m = new InfoDialog(myFrame, "Please select a Form to save.", true);
                m.show();
                if (m.isOK()) {}
            }
            else {
                saveForm(formID);
            }
        }
        if (arg.equals("Copy Form")) {
            if (Global.isListEmpty()) {
                return;
            }
            formsListGui.showList();
            String formID = Global.getCurrentListItem();
            if (formID == null) {
                InfoDialog m = new InfoDialog(myFrame, "Please select a Form to copy.", true);
                m.show();
                if (m.isOK()) {}
            }
            else {
                System.out.println("sending request for " + formID + " to be copied");
                LispTalker.send("(gui-copy-form \"" + formID + "\")");
            }
        }
    }//end ActionPerformed
    /**
    *Given: a string of the form ID
    *grabs the important info of directory and if all kosher, sends the info to
    *the form in question to handle flushing duties and lisp request.
    */
    private void saveForm(String formID) {
        Frame dialogFrame = new Frame();
        FileDialog dialog =
                    new FileDialog(dialogFrame, "Save a File",
                                        FileDialog.SAVE);
        dialog.setDirectory("/nfs/spectre/u4/hackenda/Forms");
        dialog.setFile(formID);
        dialog.show();
        //getDirectory();
        //getFile() and check if currently in formTable
        String directory = dialog.getDirectory();
        String form = dialog.getFile();
        //send directory to saveForm function if form exists
        DataSimpleForm tempForm =
                                    (DataSimpleForm)Global.FormTableFind(form);
        if (tempForm != null) {
            //saveForm is in DataSimpleForm where you can find the LispTalker.send
            //message which currently is commented out.
            tempForm.saveForm(directory);
        }
        else {
            InfoDialog m = new InfoDialog(myFrame, form + " does not exist.", true);
            m.show();
            if  (m.isOK())  {}
        }
    }
}//end Forms/3
```

Appendix C

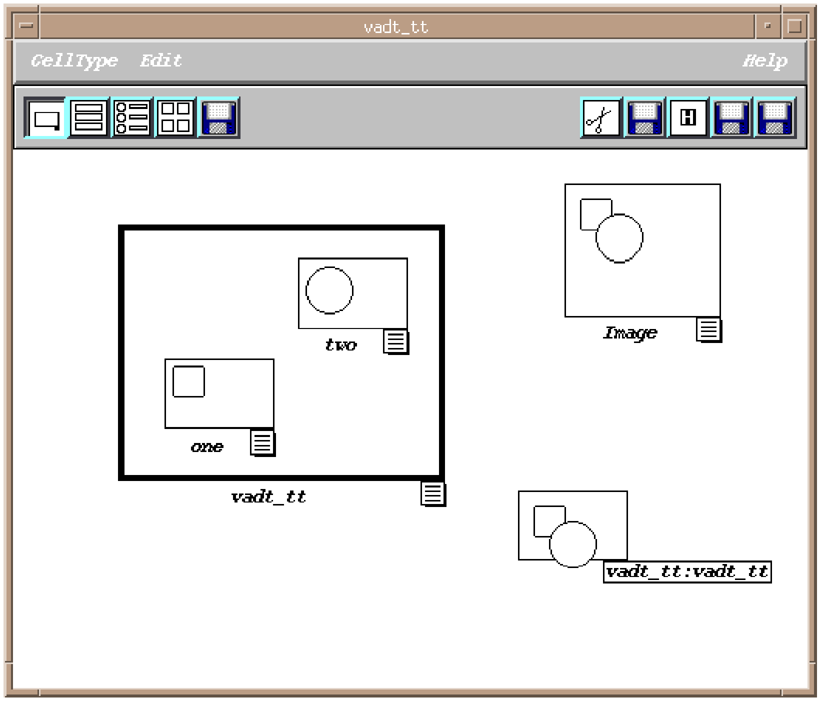
Pictures of Java Gui



Picture of the main form, the forms list, two simple forms with simple cells showing arrow dependencies.



Picture of a simple form containing the four major types of ROs with arrows.



Picture of a Vadt form with the Absbox, Image cell and a simple cell whose posted formula has a reference to the Absbox.

Appendix D

<u>**Keywords used in socket messages returned from Lisp**</u>

- **DONE**: informs Java code that the files for the Lisp implementation have finished loading and the dialog box requesting the user to wait is removed from the screen.

- **WINDOW:** informs Java code that the information in the message is a form and its ROs, either default, loaded from disk, or a copy of a form.

- **VALUE:** the following message is an evaluated formula (a value) to assign an RO.

- **FMLA-ARROWS-LIST:** the following message is a list of all ROs that the accompanying cellID affects.

- **HELP:** the following message is a string representing a help file for a specified form.

- **ADDCELL:** the following message contains information to create a default cell of the type requested.

- **CUTCELL:** the following message informs Java that a requested cell has been removed from the Lisp model and Java should also delete it.

- **NAMECELL:** a requested name has been assigned and Java should also cache and display the name.

- **ATTRIBUTES:** a requested change of a cell's attributes has been made on the Lisp side and Java should follow suit.

- **UNLOADFORM:** Lisp has removed a requested form from the underlying model and the same should be done by the Java side.

- **MESSAGE:** a message sent to the Java side and intended for a display in a Java dialog box, such as a caught exception.

- **OPTIONS:** the new state of the options list for a requested change in the list for an option or radio cell.

7. References

[Alexander et al. 1977] Alexander, Christopher, S. Ishikawa and M. Silverstein, A Pattern Language, Oxford University Press, 1977.

[Booch 1996] Booch, Grady, The Best of Booch/Grady Booch, Ed Eykholt, editor., SIGS Books and Multimedia, Prentiss Hall Inc., 1996.

[Bourne 1992] Bourne, John R., Object-Oriented Engineering: Building Engineering Systems Using Smalltalk-80, Richard D. Irwin, Inc., and Aksen Associates, Inc., 1992.

[Burnett 1991] Burnett, Margaret, Abstraction in the Demand-Driven, Temporal-Assignment, Visual Language Model. Ph.D. Thesis, University of Kansas Computer Science Department, August, 1991.

[Burnett 1993] Burnett Margaret, "Types and Type Inference in a Visual Programming Language", 1993 IEEE Symposium on Visual Languages, Bergen, Norway, Aug. 24-27, 1993, pp238-243.

[Burnett and Ambler 1992] Burnett, Margaret, Allen Ambler, "A Declarative Approach to Event-Handling in Visual Proggramming Languages", 1992 IEEE Workshop on Visual Languages, Seattle, Sept. 1992, pp. 34-40.

[Burnett and Ambler 1994] Burnett, Margaret M., Allen L. Ambler, "Interactive Visual Data Abstraction in a Declarative Visual Programming Language", Journal of Visual Languages and Computing, Vol. 5, No. 1, March 1994, pp. 29-60.

[Burnett et al. 1995] Burnett, M., Goldberg A., Lewis, T., editors. "What is Visual Object-Oriented Programming?" in *Visual Object-Oriented Programming: Concepts and Environments*., Prentis-Hall/Manning Pubs., 1995.

[Collins 1995] Collins, Dave, Designing Object-Oriented User Interfaces, Benjamin/Cummings Publishing Company, Inc., 1995.

[Eckel 1998] Eckel, Bruce, Thinking in Java: The definitive Introduction to Object-Oriented Programming in the Language of the World-Wide Web, Prentiss Hall Inc., 1998.

[Eliens 1995] Eliens, Anton, Principles of Object-Oriented Software Development, Addison-Wesley Publishing Company Inc., 1995.

[Gamma et al. 1995] Gamma, E., R. Helm, R. Johnson, and J. Vlisssides, Design Patterns, Addison-Wesely Publishing Company, Inc., 1995.

[Krasner and Pope 1988] Krasner, Glenn, and Stephen Pope, “A Cookbook for using the model view controller paradigm in Smalltalk-80”, Journal of Object-Oriented Programming, Vol. 1, #3, pp. 26-49, 1988.

[Lee 1993] Lee, Geoff, Object-Oriented GUI Application Development, Prentiss Hall, Inc., 1993.

[Martin et al. 1997] Martin, Robert C., James W. Newkirk, and Bhama Rao, "TaskMaster: An Architecture Pattern for Gui Applications" in C++ Report, March 1997.

[Myers 1993] Myers, Brad A., "Why are Human-Computer Interfaces Difficult to Design and Implement?". A technical paper from the Computer Science Department, Carnegie Mellon University, number CMU-CS-93-183, July 1993.

[Myers et al. 1990] Myers, Brad A., Dario A. Giuse, Brad Vander Zanden. "Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods" in Proceedings of OOPSLA'92: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Oct. 18-22, 1992.

[Myers et al. 1990] Myers, Brad A., Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Micklish, and Phillippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces". IEEE Computer 23, 11 (Nov. 1990), pp. 71-85.