ENHANCEMENTS OF THE GENERIC MANIFOLD USER INTERFACE

By

RAGHAVENDRA Y. SIDHANTI

A thesis submitted to the

Graduate School-New Brunswick

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Master of Science

Graduate Program in Electrical & Computer Engineering

Written under the direction of

Dr. Ivan Marsic

And approved by

_____

_____

_____

New Brunswick, New Jersey

October, 2009

ABSTRACT OF THE THESIS

Enhancements of The Generic Manifold User Interface

By

RAGHAVENDRA Y. SIDHANTI

Thesis Director:
Dr. Ivan Marsic

Manifold is an attempt to create a generic UI which would be application-independent, where the UI can be easily "detached" from one application and "attached" to another one. The *Model-View-Controller (MVC)* design pattern is a popular design pattern used in User Interfaces which has been employed in manifold. In this pattern, a user generates input device events which are interpreted as actions on the domain *model* via a Controller. After execution of the requested actions, the model sends notifications about the effect of the actions, and the notifications are visualized as feedback to the user.

Different applications have different sets of inputs and so the UI should be able to translate a users input on manifold's workspace to that of the domain model. To solve these issues, the MVC design and the Event Frame concept has been incorporated. The `EventFrame` conveys user's intentions (event) in a standardized format to the `Controller` to be acted up on the domain model.

The `EventFrame` being the only "medium" to communicate to the `Controller` makes it pivotal in communication between the UI and the domain model. We envision manifold to grow in to a complex application that would cater to various complex domain models on the internet where XML is gaining popularity as a way to

share and transport data. In such a situation, the number of messages sent between the `manipulator` and `Controller` would be enormous. We felt that in such a situation, the Event Frame would be a bottle neck to performance.

Considerable amount of my work in this thesis concerns with re-engineering the Event Frame in order to make it "web friendly". We eliminated the `Hashtable` to represent the `EventFrame` with a *comma* separated `String` that would make it easier for the XML parser to parse data. The modifications made to the application and performance enhancements have been described in detail.

Property editors are used to edit the properties/attributes of a selected glyph. It exposes the glyph's properties for editing. Every time a new glyph is selected, the old editors are emptied from the viewer, and the new set of editors are loaded. My work in this thesis describes the implementation of newer property editors incorporated in to the property viewer panel that will provide a user with enhanced options to edit a selected glyph. The newer property editors incorporated are a fill color editor which fills the interior of a glyph with a user specified color and a stroke editor which edits the stroke of a glyphs boundary.

Lastly, while the basic feature of being able to draw a glyph and perform actions on them using tools worked, there were certain issues with the *property viewer* that prevented the *properties editors* from being displayed in the *property viewer* pane. My preliminary work was to eliminate these issues. I have described the issues with in the application and the solutions employed to eliminate them.

Table of Contents

List of illustrations

Chapter 1

Introduction

Most of us are very familiar with the usage of a remote control to operate a television, a dialing pad to make a telephone call and a web browser to browse the internet. These are devices that make life convenient by making things easily accessible. These devices essentially provide a medium through which a user can interact with an underlying system that may be more complex than its operation. Moreover these are devices that are specifically built for a purpose and are solely operated to achieve this purpose.

Essentially, the above mentioned devices are *interfaces* that provide means to interact with something. While *user interfaces* (1) provides means by which a person can interact with a system – a particular machine, a computer program, a device. It provides an *input* through which the user can manipulate a system and *output* allowing the system to notify the user of the requested changes. A *graphical user interface (GUI)* (2) in particular embodies the basic principles of a user interface by providing graphical icons, visual indicators and visual outputs. A GUI is a human-computer interface that uses windows, icons and menus and which can be manipulate by a mouse (or other pointing devices) and often to a limited extent by a keyboard as well. It stands in sharp contrast to the command line interface which uses only text and can be operated solely by a keyboard.

Here, my work mainly focuses on GUI's. Using the user interface, the user can:

- Modify the properties of model elements
- Select the viewpoint and navigate the "model world"

Generally, UI's are molded about its particular application domain hence requiring a great deal of work if it were to be molded around a different application. This is particularly true for interfaces based on hand operation of input devices. To overcome this problem Dr. Ivan Marsic and his team presented a design called *Manifold* (3). Manifold is an attempt to create a generic UI which would be application-independent, where the UI can be easily "detached" from one application and "attached" to another one. The first version of Manifold appeared in (4). And, was also based on the work (5; 6). My work has been to extend the work initiated by Dr. Marsic in many ways.

My work encompasses the following areas,

- Debugging *errors* existent in manifold
- Addition of *Property Editors*
- Re-engineering the *Event Frame*

In order to understand the problems and the solutions employed to solve them, it is critical to understand the architecture of manifold. This introduction provides an over view of the Manifold frame work. The Chapters that follow will provide a serious, detailed explanation of these problems and the solutions employed.

1.1 Manifold Framework: Current Implementation

In order to understand the manifold framework, it is important to visualize the *domain model*. **Figure 1** shows the use case diagram of the application (domain model) that will be used as an example for employing the Manifold framework.

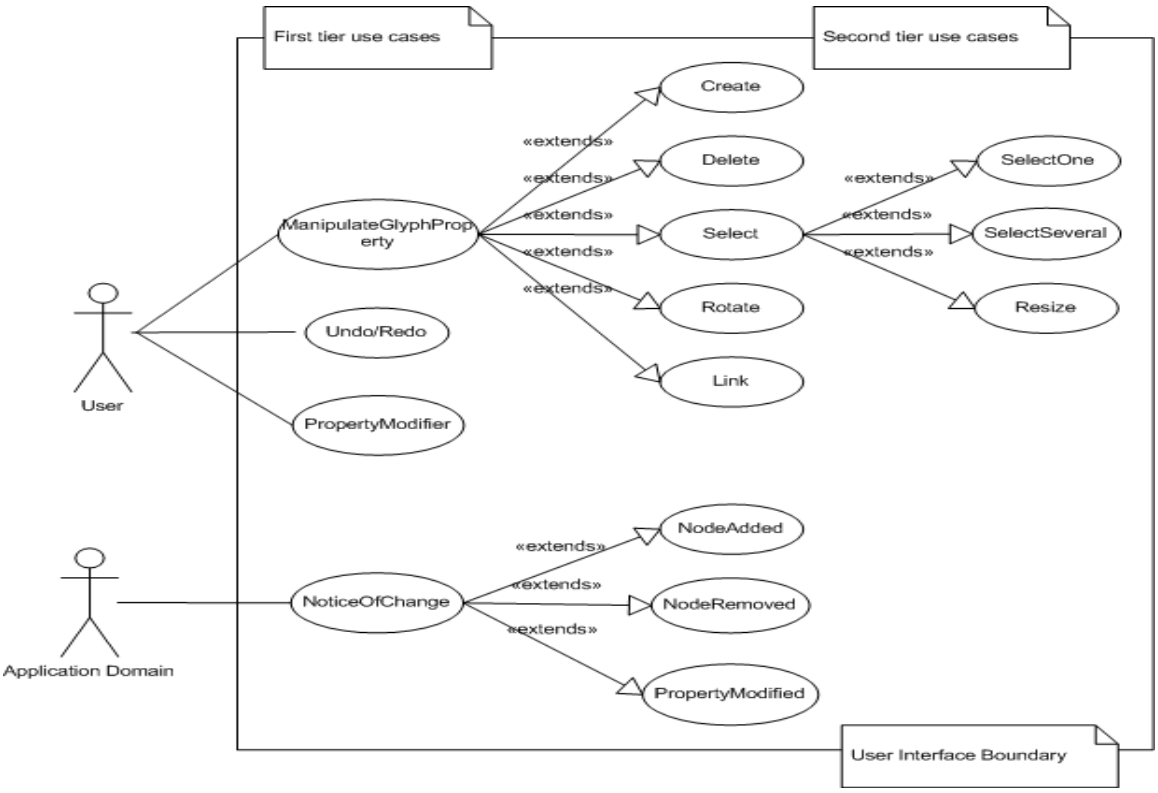

Figure 1: Use Case diagram for the example application of the manifold framework

To provide a visual appearance of the underlying elements of the domain model, glyphs have been used. A *Glyph* (7) is a visual representation corresponding to a model data element in a domain model. It visualizes the domain model's state changes. The name "glyph" is borrowed from typography to connote simple, lightweight objects with an instant specific appearance. The key purpose of *Glyph* is to implement composite design pattern, so to hierarchically compose glyphs into more complex figures.

Figure 2 displays a prototype of the current implementation of Manifold. Using this interface one can create, delete, select, rotate and link various glyphs. It is also possible to attribute various properties to these Glyphs through a *Property Editor*.

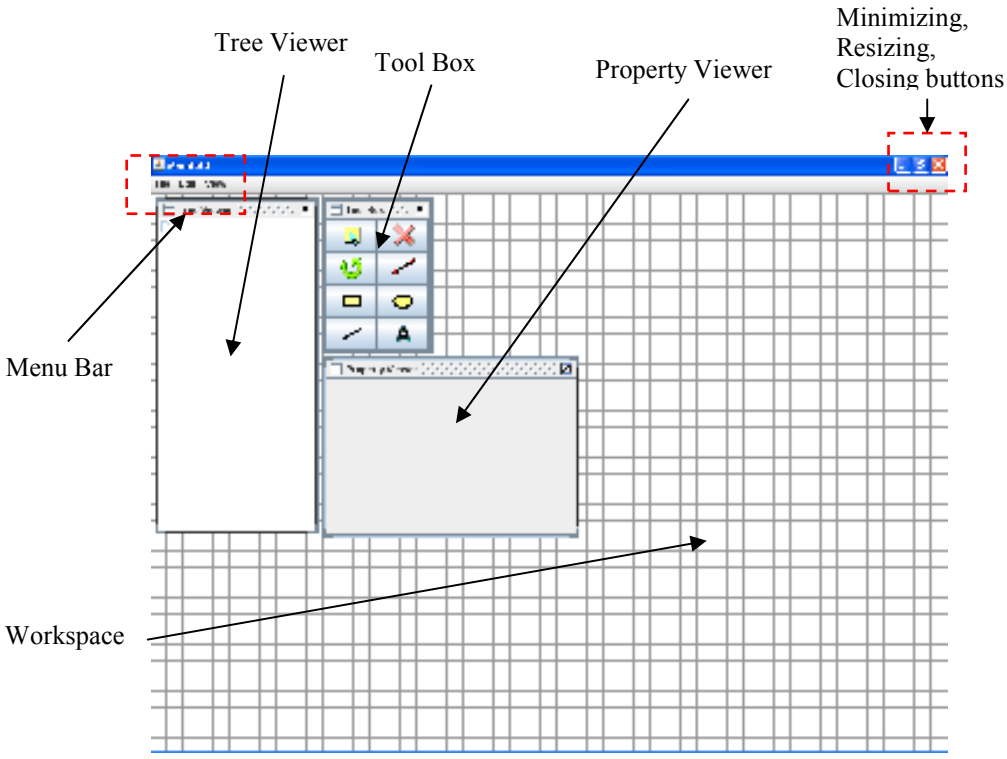

Figure 2: Manifold User Interface

Features incorporated in the prototype Manifold user interface include:

- A *Workspace*, which is the area with in the interface where a user can create glyphs such as rectangles, ellipses, lines and text boxes. With in the *Workspace*, the glyphs can be selected, rotated, linked and deleted. Various property attributes can be assigned to these glyphs with in the *Workspace* using the property editors provided in the *Property Viewer* panel.
- A *Tree Viewer* provides the ability to view the existing glyphs in a tree format. The tree is arranged on the basis of the order in which the nodes of the tree (Glyphs) were created in the workspace. Each time a glyph is created in the workspace, a glyph will be automatically added to the tree in the *Tree Viewer*.

Similarly, deletion of a glyph in the workspace will automatically result in deletion of a node in the *Tree Viewer*.

- *Property Viewer* allows the user to set certain attributes to the glyphs created in the workspace via a set of property editors. The property editors in the *Property Viewer* are visible when a glyph is created or an existing glyph is selected. The properties incorporated in to the *Property Viewer* of the interface include; a *line color* which defines the boundary color through a color palette, five pre-defined *stroke types* that define the type of stroke and *width* of these strokes for the rectangle, ellipse and line glyph. While the rectangle and ellipse have an additional property called *fill color* that defines the fill color of these closed glyphs through a color palette.
- *Tool Box* provides the user a set of buttons to perform various functionalities in the workspace. There are buttons for *Creating* a glyphs (rectangles, ellipses, lines) and text box (under development). Buttons for: a *selector* for selecting the glyphs, *delete* to delete a glyph, *rotator* to rotate a glyph and a *linker* to create links between two glyphs (under development).
- A *Menu Bar* which provides access to several advanced features such as linking two or more interfaces over a network, Saving and loading files, ability to open multiple interfaces and editing the properties of the interface is under development.

Manifold user interface is a work in progress. It employs Sun Microsystems Java Technology as its primary coding language to design the interface and its underlying

application. My work on the interface has been on developing some of the features of the application which makes the system a stable platform to work on. My work has been outlined in the next section.

1.2 Brief Introduction to my work

My work on manifold was on three specific areas:

1.2.1 Property Viewer

When I began working on Manifold, there were several issues with in the application. These issues prevented the rendering of individual *Property Editors* pertaining to a glyph in the *Property Viewer* panel. My primary task was to debug these issues and make the *Property Viewer* a functional entity of the interface.

1.2.2 Property Editors

I worked on incorporating new *Property Editors* for the glyphs with in the *Property Viewer* of Manifold.

1.2.3 Event Frame

An *Event Frame* transcribes user's actions in to a format that can be used to communicate with the *domain model*. I worked on re-engineering the Event Frame by replacing its format from a *hash tables* to a *string*. The idea behind the change was to make the application more viable over the internet.

Chapter 2

Architecture Overview

The design of Manifold may seem very intuitive at the beginning. But, as one drill down in to the code, the ideas and techniques employed may not seem all that intuitive. In order to better understand my work it is important that one understands a general overview of its architecture. In this section, I have described certain critical aspects of the architecture and provided a "some what" in depth analysis of the topics that would be needed in understanding my work. For further clarifications and a better description of the Manifold architecture I recommend the documentation provided by Dr. Marsic which I have cited at.

2.1 Model-View-Controller Design Pattern

The *Model-View-Controller (MVC)* design pattern is a popular design pattern used in User Interfaces (8; 9; 10). The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts: the *model*, the *view*, and the *controller*. The *model* represents information (the data, underlying application) of the application; the *view* corresponds to elements of the user interface such as workspace, text, checkbox items, and so forth; and the *controller* manages the communication of data and the business rules used to manipulate the data to and from the domain model. **Figure 3** illustrates an abstraction of the user interface using the MVC model.

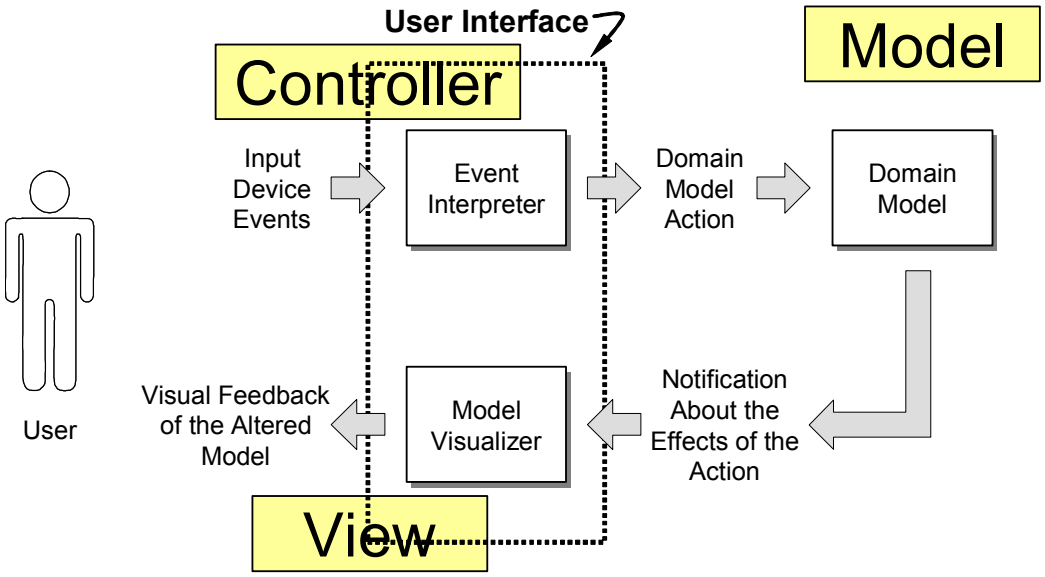

Figure 3: Abstraction of the Model-View-Controller (MVC) Design Pattern *(Marsic, I)*

In this model a user generates an *Input Device Event* which is interpreted as actions on the domain model by the interface. These events are converted to Domain Model Action. After executing the requested actions, the model sends notification about the effects of the actions in the reverse order and the notifications are visualized as feedback to the user. This is how classical *Observer* design pattern works, i.e., the Observer reads the Subject state upon being notified about the state changes (9). Conversely, in the Java event delegation pattern (11), the event source sends to the listener the event containing the new state information along with the notification.

2.2 Model

Domain Logic represents the calculations and data storage that form the core of an application. The *model* encapsulates the functional core of the application which is its *Domain Logic*. The goal of MVC is to make the model independent of the *view* and

controller which together form the user interface of the application. An object may act as the model for more than one MVC triad at a time.

Since the *model* must be independent, it cannot refer to either the *view* or *controller* portions of the application. The model may not hold direct instance variables that refer to the *view* or the *controller*. It passively supplies its services and data to the other layers of the application.

2.2.1 Domain Model Visualization

In manifold, Glyphs have been used to visualize the elements of the domain model. *Glyph* is a visual representation corresponding to a model data element in a domain model. It visualizes the model's state changes. The key purpose of Glyph is to implement the *Composite* design pattern (9), so to be able to hierarchically compose glyphs into more complex figures.

The base class for glyphs is `manifold.Glyph`, which is an abstract class, see **Figure 4**. The class `manifold.impl2D.Glyph2D` implements geometric functionality specific for Java 2D (12) domain on the Glyphs. From this, two types of two-dimensional glyphs are derived:

1. *Leaf* glyphs, which have visual appearance, i.e., they can be rendered. For example, glyphs for primitive geometric shapes inherit from `manifold.impl2D.GeometricFigure`, and are implemented in the package `manifold.impl2D.glyphs`.

2. *Inner* glyphs represented by `manifold.impl2D`.TransformGroup, which is composite, a container for groups of glyphs.

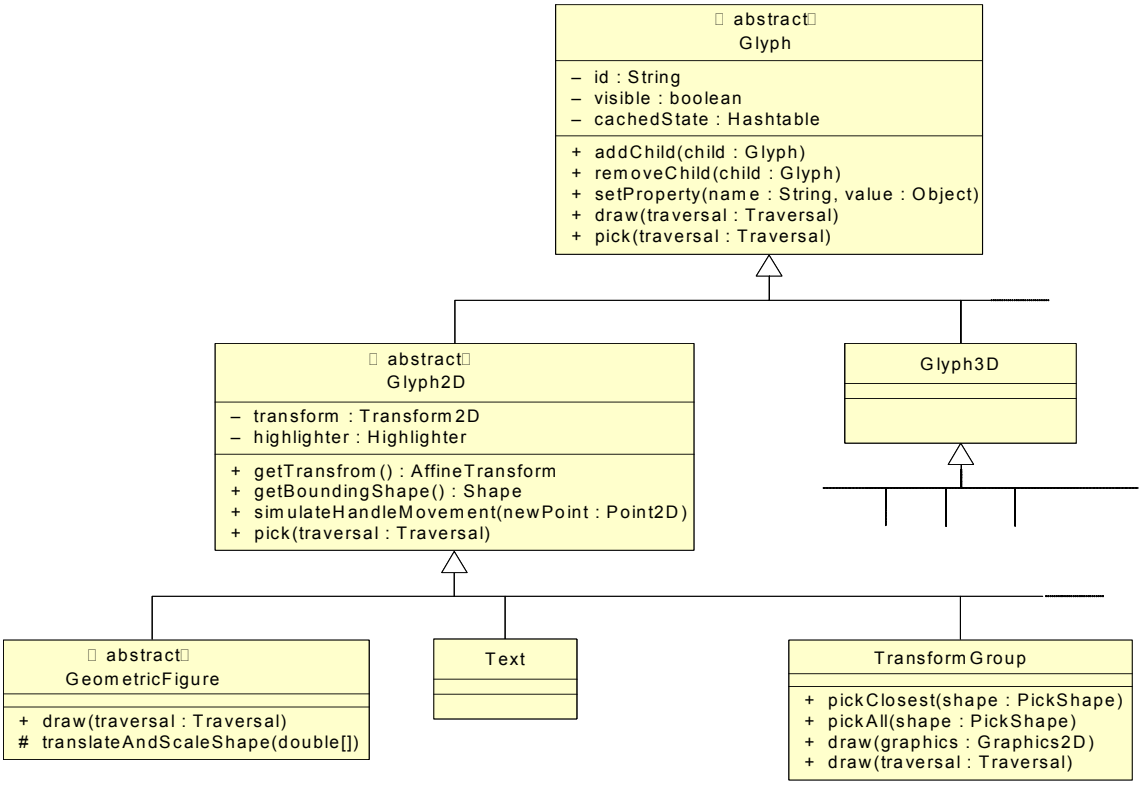

Figure 4: UML class diagram of glyph inheritance hierarchy in Manifold (Marsic, I)

2.3 Controller

A strict distinction has been maintained between the domain model and UI. This is critical in order to create an effective UI. A *controller* is the means by which the user interacts with the application. A *controller* accepts input from the user and instructs the *model* and *view* to perform actions based on that input. In effect, the *controller* is responsible for mapping end-user action to application response. For example, if the user clicks the mouse button or chooses a menu item, the controller is responsible for determining how the application should respond.

2.3.1 Parsing Input Event Sequences

When a user handles an input device(s), it generates interaction events, which are translated to actions on the domain model. For example, the user's activity of depressing the mouse button and dragging it around the workspace has different meaning, depending on the currently selected tool. Examples are rotation of a graphical figure, resizing, translation, etc. The selected tool "knows" which one of these is currently in effect. The design espoused here is inspired by Unidraw (13; 14) and Fresco (15; 16).

2.3.2 Manipulator

To carry out the manipulation, a tool creates a "*Manipulator*". In other words, the tool encapsulates *state* and Manipulator encapsulates *behavior*. A new Manipulator object is instantiated (by invoking `Tool.createManipulator()`) at the moment the user starts a new interaction cycle and disposed of at the end of the interaction cycle. An example of "interaction cycle" is: (1) user depresses a mouse button; (2) drags the mouse across the workspace; and, (3) releases the mouse button.

Roughly speaking, the tool encapsulates the static part of the interpretation apparatus, i.e., describing *what* this tool does. Manipulator encapsulates the dynamic part, the transient state associated with a single manipulation cycle.

The manipulator plays a key role in orchestrating the event interpretation. Once an event is created, the events are transcribed in to an *EventFrame* and notified to the *Controller*.

2.3.3 Interaction with the domain model

In manifold, *Controller* is a single object acting as a gateway between the presentation and domain modules of the system. Conversely, in the MVC design pattern, *Controller* is a component of the pattern, usually implemented as a set of cooperating objects working together on the input interpretation task.

It is common to think of the application domain (also called functional core, or, application logic) as not dealing with user interface issues. True, the domain does not deal with the presentation of information to the user and other aspects of interaction. Nonetheless, it is on the way of the data flow and its functioning becomes apparent in the interaction.

The domain module may not be "aware" of the user, but the user is keenly aware of the domain (via the presentation). Therefore, the domain designer may need to take into account the impact of design decisions on the efficiency/effectiveness of interaction. It is noteworthy that the event frames do not contain explicit information about the current operating mode of the user activity. For example, regardless of whether the operation is rotation or scaling or translation, the event frame only contains the glyph identity and its new *transformation* attribute. If it for some reason needs to know what transformation is being applied, it is possible to decompose the transformation into constituent "pure" transformations by invoking

2.4 View

The *View* is responsible for mapping graphics onto a device. A view typically has a one to one correspondence with the display surface and knows how to render to it. A

view attaches to a model and renders its contents to the display surface. In addition when the model changes, the view automatically redraws the affected part of the image to reflect those changes. There can be multiple viewports onto the same model and each of these viewports can render the contents of the model to a different display surface.

Chapter 3

Property Viewer

As mentioned earlier, my work on manifold is an extension of Dr. Marsic's. During my early encounters with Manifold I noticed a lot of errors being generated while running the application. While the basic feature of being able to draw a glyph and perform actions on them using tools worked there were certain bugs with the *property viewer* that prevented the *properties editors* from being displayed in the *property viewer* pane. My preliminary work was to eliminate these bugs.

3.1 Introduction: Property Viewer

The design of manifold is very elegant. Inter coupling of various domains of the interface have been minimized to a great extent. What I mean is that, a glyph can be drawn on the workspace even if the *property viewer* doesn't function or the *tree viewer* does not function. A strict de-coupling of the features has been ensured through out the design which is what makes it very interesting. Moreover, it makes the code easier to read and to de-bug.

The role of *property viewer* is to display editable properties (attributes) of the glyphs in the workspace in the form of *property editors*. *Property Viewer* displays only *one* glyph at a time—the one that is selected. When a glyph in the workspace is selected using the *selector tool* its corresponding properties are displayed on the *property viewer.*

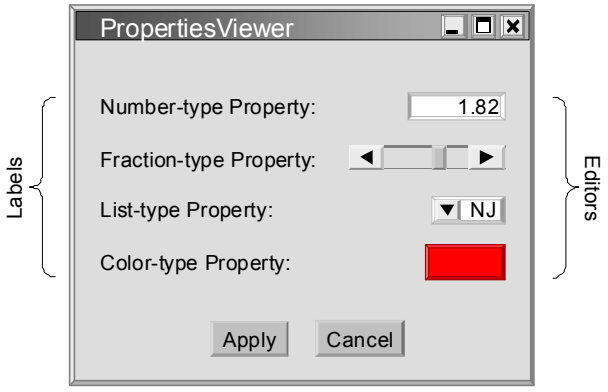

Figure 5: Example of a property editing dialog box. Property editors allow editing the property values. (Marsic, I)

Similarly, when a new glyph is created on the workspace using the *creator* tools, its properties are automatically displayed on the *property viewer* panel. This is because when a new glyph is being created, it is the currently selected glyph.

The properties that could be displayed can be of several different types. For example, a *rectangle* or an *ellipse* glyph could have properties that could help set their outer boundary and fill colors. They could have options to decorate their boundary lines with decorative lines (dashed lines, dotted lines, dash-dotted lines etc). Have options to erase part of the boundaries or modify their shapes. The possibility of *zooming in* or *zooming out* of figures. And, the possibility of placing a selected glyph in the *background* or *foreground* when several glyphs are present. Some of these properties could be applied to *line* glyphs as well. Other non glyphs like *text box* can benefit from the *property viewer*, where one can vary *font type* and *size*. The ability to add 3D art in fonts and draw tables are some other advanced features that a *properties viewer* can provide.

It can be noticed that the number of different properties that can be created in the *property viewer* and attributed to a glyph are plenty and can be left to the imagination of the developer. As mentioned above, there is only one property viewer instantiated per application. Every time a new glyph is selected, the old editors are emptied from the

viewer, and the new set of editors are loaded. **Figure 5** illustrates the idea of *property viewer* and its *editors*.

3.2 The Issue: Property Viewer Issues

Up one running the code for the first time, I noticed that there was a serious problem with interface. Several error messages were being generated at the start of the program and the *property viewer* panel wouldn't work! **Figure 6** displays the errors generated at the start of the application. Each time a glyph was created or selected two dummy buttons would show up on the *property viewer* panel which up on clicking would not change the attributes of the glyphs.

```
java.lang.NoSuchMethodException: <unbound>=PropertyEditorsPanel.add(FontEditor);
Continuing ...
java.lang.ClassCastException: manifold.swing.PropertyEditorsPanel cannot be cast to manifold.PropertyEditor
        at manifold.swing.PropertiesViewer.buildLUT(PropertiesViewer.java:317)
        at manifold.swing.PropertiesViewer.setFileName(PropertiesViewer.java:296)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at java.lang.reflect.Method.invoke(Unknown Source)
        at sun.reflect.misc.Trampoline.invoke(Unknown Source)
        at sun.reflect.GeneratedMethodAccessor1.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at java.lang.reflect.Method.invoke(Unknown Source)
        at sun.reflect.misc.MethodUtil.invoke(Unknown Source)
        at java.beans.Statement.invoke(Unknown Source)
        at java.beans.Expression.getValue(Unknown Source)
        at com.sun.beans.MutableExpression.getValue(Unknown Source)
        at com.sun.beans.ObjectHandler.getValue(Unknown Source)
        at com.sun.beans.ObjectHandler.endElement(Unknown Source)
        at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.endElement(Unknown Source)
        at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanEndElement(Unknown Source)
        at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl$FragmentContentDriver.next(Unknown
        at com.sun.org.apache.xerces.internal.impl.XMLDocumentScannerImpl.next(Unknown Source)
        at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanDocument(Unknown Source)
        at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(Unknown Source)
        at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(Unknown Source)
        at com.sun.org.apache.xerces.internal.parsers.XMLParser.parse(Unknown Source)
        at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.parse(Unknown Source)
        at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser.parse(Unknown Source)
        at javax.xml.parsers.SAXParser.parse(Unknown Source)
        at javax.xml.parsers.SAXParser.parse(Unknown Source)
        at java.beans.XMLDecoder.getHandler(Unknown Source)
        at java.beans.XMLDecoder.readObject(Unknown Source)
```

Figure 6: Errors generated on the first run

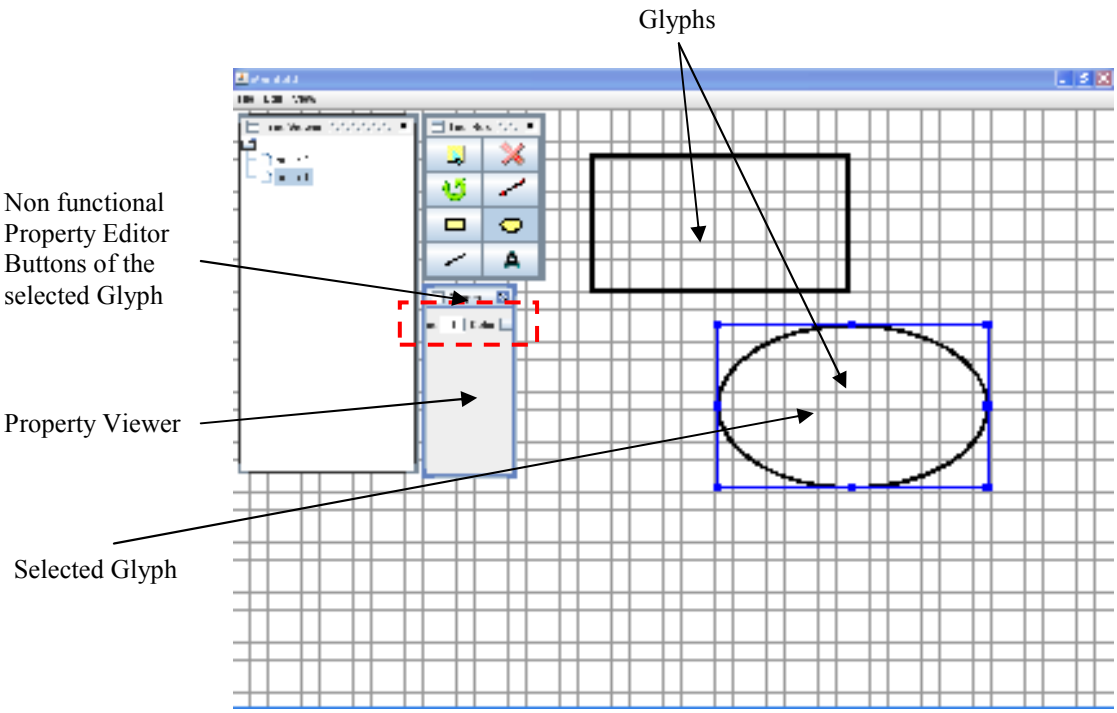

Figure 7: View of Manifold on its initial runs. It had a non functional property editor panel on selection of a glyph. Notice that the aesthetics of the property viewer weren't great too.

Figure 7 displays the initial version of the user interface with non functional buttons. My first task was to get the *property viewer* up and running.

3.3 Design: Property Viewer

The java code `manifold.swing.PropertiesViewer` is responsible for displaying the *property editor* panels in the *property viewer* pane. Its implementation is shown in **Figure 8**. The glyph-specific *property editors* are contained in the `manifold.swing.PropertyEditorsPanel`, which is specific to different glyph types and is re-loaded every time a new glyph is selected. `PropertyEditorsPanel`

contains multiple property editors, which are subclasses of `javax.swing.JComponent (17)`.

Note that the information about the editable properties is known only to the glyph's `PropertyEditorsPanel`, not to the glyph and not to the corresponding domain node, so the property viewer in fact inquires the `PropertyEditorsPanel`.

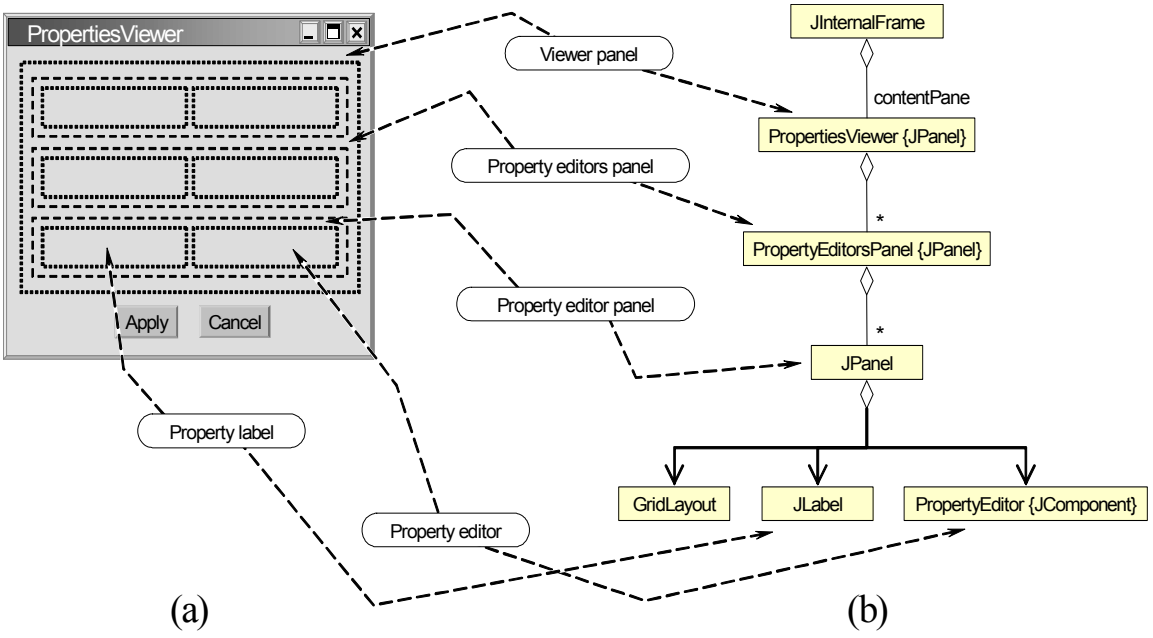

Figure 8: (a) A dummy Implementation of the Properties Viewer (b) Class hierarchy of the Properties viewer. (Marsic, I)

The `PropertyEditorsPanel` is a subclass of `javax.swing.JPanel (18)`. With in the `PropertyEditorsPanel` are a set of smaller `JPanel`'s with a *Grid Layout* of 1x2 as shown in **Figure 9**. Each of this subset (1x2) `JPanels` hold the editable properties of a selected glyph. The left box (i.e [1,1]) contains a `javax.swing.JLabel (19)` which displays the *name* of the editable property and

right box (i.e. [1,2]) contains the editable properties which are `javax.swing.JComponent`'s.

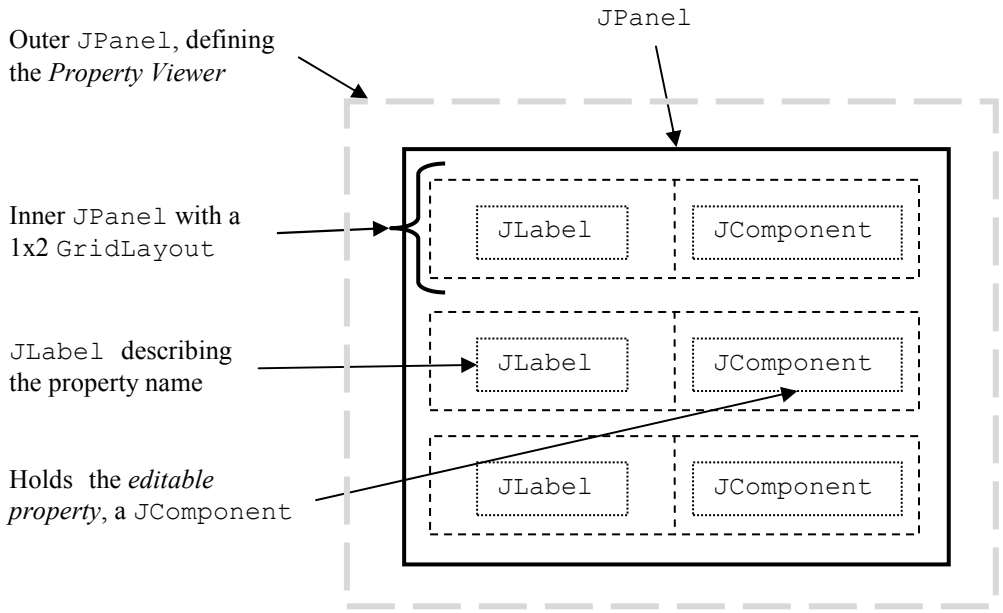

Figure 9: Implementation of `manifold.swing.PropertyEditorsPanel`

A `PropertyEditorsPanel` containing various *editable properties* and their *label* is specific to a particular glyph. Hence, each glyph has one such `PropertyEditorsPanel` containing all its *editable properties* in it. These panels and their corresponding glyph *names* are held in a *hash table* created via an XML file `editors.xml`. The `editors.xml` file also lists the *editable properties* to be included in the `PropertyEditorsPanel` for a particular glyph.

Once the `PropertyEditorsPanel` is created, its contents are displayed via a *protected* method `buildLUT()` in `manifold.swing.PropertiesViewer`. `buildLUT()` builds a look up table containing these editable properties. Further more, it *de-couples* the `PropertyEditorsPanel` to obtain the `JComponent` (i.e the

editable properties) and assigns them to a generic interface `manifold.ProperyEditor`. Through the `PropertyEditor` interface the properties are subsequently altered.

The reason behind the *editable properties* not functioning in the *properties viewer* was because this *de-coupling* and its subsequent assignment to the `ProperyEditor` wasn't taking place in a *proper* fashion. By *proper,* I mean in the opposite way to which the `PropertyEditorPanel` was created.

3.4 Solution: Issue 1

Figure 9 displays the design of the `PropertyEditorPanel.` And its 1x2 inner `JPanel` for each *editable property*, and a `JLabel` and `JComponent`. In order to successfully assign the `JComponent` to the interface `manifold.PropertyEditor` the following steps have to be followed sequentially,

1. Retrieve the *values* of the *hash table* using a Java `Iterator (20)`. Each *value* corresponds to a `PropertyEditorsPanel`.
2. On each of these `PropertyEditorPanel`'s, use a loop to obtain each of the inner `JPanel`'s containing the `JLabel` and `JComponent`.
3. From each of these inner `JPanel`'s, retrieve the `JComponent` and assign it to the interface `PropertyEditor` after proper *type casting*.

It should be noted that, there are three loops placed in a hierarchy in order to make a successful assignment to the `PropertyEditor`. The original code ***wasn't*** following this sequence and hence the cause of errors.

The original code stub which caused the error was,

```
for ( Iterator i_ = editorPanels.values().iterator(); i_.hasNext();){

      PropertyEditor editor_ = (PropertyEditor) i_.next();
      editor_.setPropertiesViewer(this);

}
```

Here the `editorPanels` is the *hash table*. In this code, the first step of the sequence existed while the other two were missing. Hence, these lead to the assignment of a `PropertyEditorPanel` to the `PropertyEditor` interface. This was an in-appropriate assignment.

In order to correct the bug, I followed the sequence as mentioned above. This resulted in a code which looks like this,

```
for (Iterator i_= editorPanels.values().iterator(); i_.hasNext();){

      JPanel newPanel_ = (JPanel) i_.next();

      for(int j_=0;j_< newPanel_.getComponentCount();j_++){

            Component comp_= ((JPanel)
            newPanel_.getComponent(j_)).getComponent(1);

            if(comp_ instanceof PropertyEditor){

                  PropertyEditor editor_ = (PropertyEditor) comp_;
                  editor_.setPropertiesViewer(this);
            }
      }
}
```

Following the code changes, I expected the application to work the way it was supposed to but, it did not. While the initial errors (**Figure 6**) on the application start up did not re-occur this time, there were a series of *new* errors generated when the properties

were assigned to a selected glyph. These errors were pertaining to a `java.lang.NullPointerException (21)`. On closer inspection of the errors and the code, I realized that the source of the error was still in `manifold.swing.PropertiesViewer`. However, this time the cause of the error was in a different method called `selectionsChange`.

3.5 Solution: Issue 2

`manifold.SelectionsListener` is an interface responsible for notifying listeners (22) about changes to the list of selected glyphs inside the workspace. The listeners are notified if more glyphs become selected or some become de-selected. Events of this type are generated by `manifold.SelectionsModel`. `selectionsChange` is a method called by `SelectionsListener` to obtain the `PropertyEditorsPanel` of the currently selected glyph by providing the *node ID* of the currently selected glyph. Through the `PropertiesEditorsPanel` the system accesses the editable properties (`JComponent's`) and the properties are assigned to the selected glyph.

The `JComponent's` belong to a package `manifold.swing.editors`. These Classes implement an `ActionListener (23)` and `PropertiesEditor`. Every time a property is assigned to a glyph through the properties viewer, an event is generated and a method `actionPerformed` is called (23). `actionPerformed` captures the users requested changes and sends the new properties to the model for re-rendering and storing in the `cachedState`. In order to perform the requested action (property change) these `JComponents` need to know the *node ID* of the node (glyph) of which the

properties need to be changed. The *node ID* has to be translated to them when the glyph is selected. `selectionsChange` being part of the properties viewer bares the responsibility of notifying these property editor classes (`JComponents`) of the current *node ID*. This wasn't being translated hence no changes were taking place to the properties of the glyph. More over, there was no defined method with in the property editor classes to translate the *node ID* to it.

I defined a method called `setCurrentNodeId` with in the property editor classes which allows other classes to communicate the current *node ID* to these classes. Code stub of `setCurrentNodeId`

```
public void setCurrentNodeId(String node_){
      currentNodeId = node_;
}
```

By adding `editor_.setCurrentNodeId(currentNodeId)` to the `selectionsChange` method in `manifold.swing.PropertiesViewer` I was able to translate the node ID of the currently selected glyph to the property editors.

Following the code changes, the *Properties Viewer* functioned as it was supposed to. With fully functional *Property Editor* buttons. **Figure 10**, shows a glyph under four different situations. **Figure 10-(a)** is a default glyph, **Figure 10-(b)** is a color glyph, **Figure 10-(c)** is a glyph with a modified width, and **Figure 10-(d)** is a glyph with modified color and width.

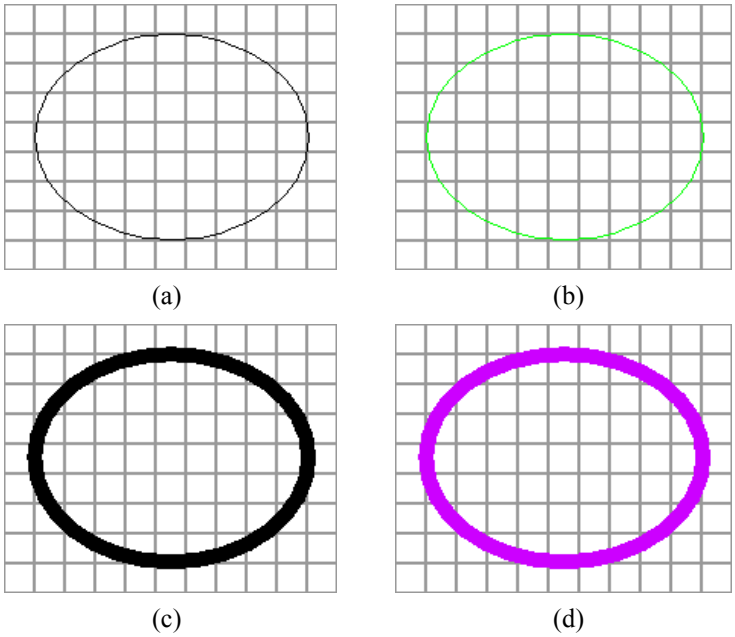

Figure 10: Functional manifold (a) default ellipse glyph (b) glyph with a colored outer line (c) glyph with a modified width (d) glyph with a modified width and color

3.6 Solution: Issue 3

The `PropertyEditorsPanel` extends a `JPanel` (18) class which provides general-purpose containers for lightweight components. Like other containers, `JPanel` uses a layout manager to position and size its components. By default, a panel's layout manager is an instance of `FlowLayout` (24), which places the panel's contents in a row. Hence, resulting in a look as shown in **Figure 11** with the property editors beside each other. In order to make the property editors appear one below the other, a `GridLayout(25)` had to be used. This was incorporated in the constructor of `manifold.swing.PropertyEditorsPanel` using the following code stub,

```
public PropertyEditorsPanel(){
      super(new GridLayout(4,1));
}
```

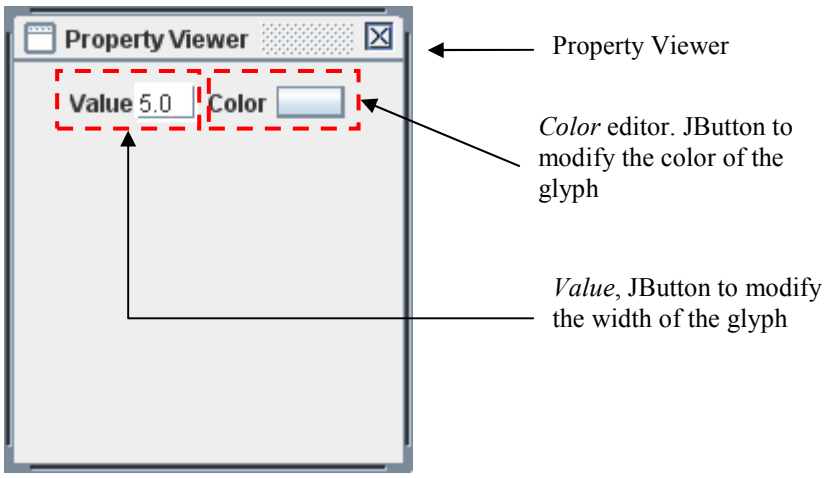

Figure 11: Property Viewer with a default Layout Manager

`GridLayout`(*x, y*) divides the panel in to *x* rows and *y* columns. In this case, 4 rows and 1 column. The number 4 is arbitrary.

Chapter 4

Property Editors

After fixing the issues, the Property Viewer could change two properties of a selected glyph (1) Change the *width* of the *Glyph* (2) Change the *color* of the *Glyph* peripheral lines. While these two properties were fully functional, the idea was to implement more editable properties of a selected glyph.

This section describes the changes made to manifold in order to avail two more editable properties namely:

1. Fill Color
2. Line Stroke

I would like to clarify a probable misconception that may exist in this section. The *property editors* that I am going to address in this section are the `JComponent`'s present in side the 1x2 `JPanel` of the `manifold.swing.PropertyEditorPanel`. This is different from the `manifold.PropertyEditor` which was talked about in the previous section. The `manifold.PropertyEditor` is an interface that is implemented by the *property editors*. In case of confusion, the best way to distinguish between the two would be by realizing the difference in font used to describe them.

4.1 Introduction: Property Editors

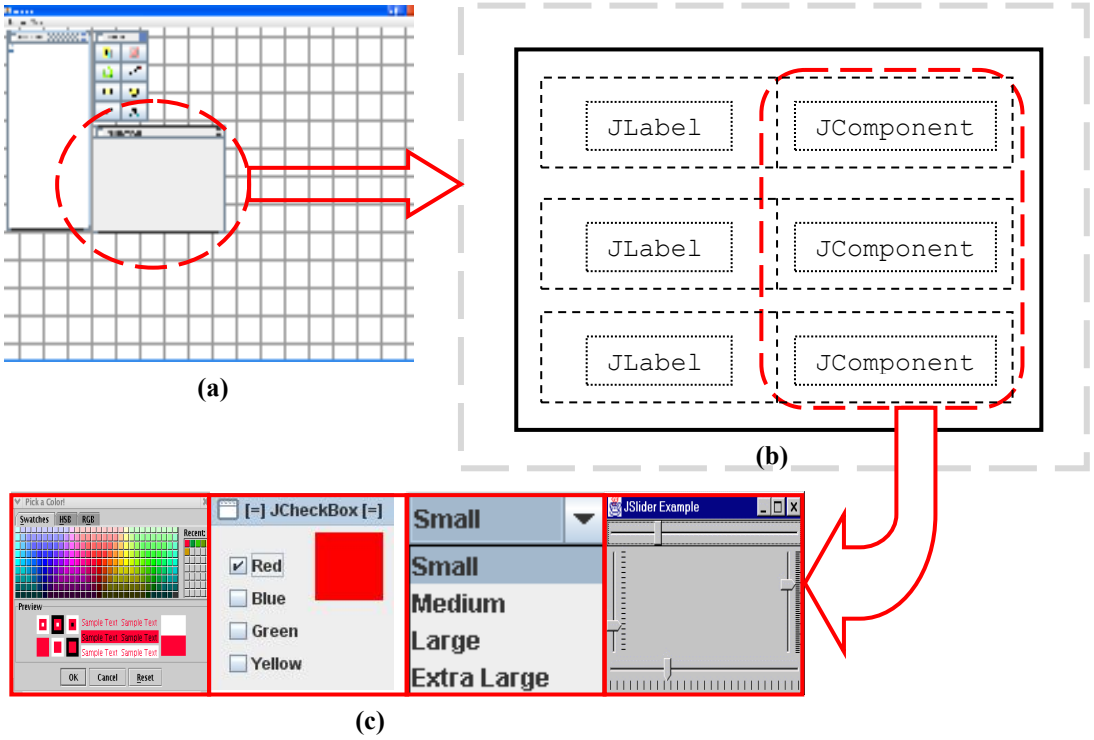

Figure 12: Illustrates the property editors. (a) The manifold user interface (b) The underlying design of the *properties viewer* with `PropertyEditorPanel` (c) Example *property editors* that could be incorporated in to the *property viewer*.

Property editors are used to edit the properties/attributes of a selected glyph. While introducing the *properties viewer* in the previous chapter, I had talked about several different properties that could be inculcated in to it. Where each of these properties would be a `JComponent` retrieved by the `PropertyEditorPanel`. These `JComponents` (property editors) could be any Java Swing Component that can directly physically alter the properties of the Glyphs or indirectly participate in it. With in manifold, these `JComponents` belong to a package `manifold.swing.editors` that contains a list of classes. Where, each class is specific to a property which would alter a glyphs

property. **Figure 12** shows what I am talking about pictorially. In **Figure 12**, (c) are some examples of *property editors* that could be incorporated in to the manifold user interface. Each property editor would be an individual class with in the package `manifold.swing.editors` which would be called by the `PropertyEditorPanel`.

4.2 Design

The package `manifold.swing.editors` contains all the property editors. Where each class files corresponds to a particular property. These classes implement a `PropertyEditor` interface and `java.awt.event.ActionListener (23)`. And generally extend a `javax.swing` (26). These classes have several Mutators for altering the values in the classes and assessors to access their values. The important ones are summarized below:

- `setValue(java.lang.Object value_)`

  Mutator for altering the value of the editable property.

- `getValue()`

  Accessor for retrieving the value of the editable property.

- `actionPerformed(java.awt.event.ActionEvent event_)`

There are certain classes of importance which have to be understood in order to understand my work. The following sections will provide a brief description of these classes and their importance.

4.2.1 manifold.swing.PropertyEditorsPanel

The `PropertyEditorsPanel` is a subclass of `javax.swing.JPanel (18)`. Which creates a set of smaller `JPanel`'s with a *Grid Layout* of 1x2 as shown in **Figure 9**. Each of this subset (1x2) `JPanels` hold the *editable properties* of a selected glyph. There is a single property editor panel object per glyph type. Each individual property editor edits a single property of the glyph. This panel holds all these editors together.

A method called `add()` with in the `PropertyEditorsPanel` is responsible for creating the smaller `JPanel`'s with a *Grid Layout* of 1x2 and placing a `javax.swing.JLabel (19)` in the left box (i.e [1,1]) which displays the *name* of the editable property and a property editor in the right box (i.e. [1,2]) contains the editable properties which is a `javax.swing.JComponent`'s.

Each of these 1x2 `JPanels` are stacked one below the other inside the `PropertyEditorsPanel`. A code stub of the `add()` method is provided below.

```
public Component add(Component component_) {

      // Set a (1-row X 2-column) grid layout.
      JPanel panel_ = new JPanel(new GridLayout(1,2));

      // Default label text is empty.
      String labelText_ = "";

      if (component_ instanceof PropertyEditor) {

      // Set the property description, if available, as the label text.
      labelText_ = ((PropertyEditor) component_).getDescription();
      }

      // Add first the text label to the grid layout.
      JLabel label_ = new JLabel(labelText_);
      panel_.add(label_);

      // Add the property value editor to the right of the text label.
```

```
        panel_.add(component_);

        // Add the entire 1x2 grid panel to the parent panel.
        return super.add(panel_);
}
```

4.2.2 editors.xml

`editors.xml` is an XML file which is part of the manifold package. Through the `editors.xml` file one can specify the set of property editors which have to be included in to the `manifold.swing.PropertyEditorsPanel`. This is done by using the `add()` method of the `PropertyEditorsPanel` where its attributes would be a property editor class from the `manifold.swing.editors` package. A `propertyName` will also be defined to aid in easy access of the property with in the application.

```
<object class="manifold.swing.PropertyEditorsPanel">
  <void method="add">
    <object class="manifold.swing.editors.ColorEditor">
      <void property="propertyName">
        <string>line.color</string>
      </void>
    </object>
  </void>
</object>
```

The above mentioned code is particular to a glyph and would result in one 1x2 inner `JPanel` embedded in to the `PropertyEditorsPanel`. This is because only one property has been specified. For more property editors, the code has to be repeated by varying the property editor's class object and value of the `propertyName`. This would result in `PropertyEditorsPanel` with several 1x2 inner `JPanels`.

Each of these `PropertyEditorsPanel's` created is specific to a glyph (eg. Rectangle, ellipse). These panels are added to a hash table through the `editors.xml` file. This hash table is accessed by `manifold.swing.PropertiesViewer` which was discussed earlier.

A code stub of the `editors.xml` file has been provided below. It is specific to a *rectangle glyph*.

```
<!-- ********* Rectangle Editor Panel ********* -->

<object class="java.util.HashMap">
  <void method="put">
   <string>rectangle</string>
   <object class="manifold.swing.PropertyEditorsPanel">
    <void method="add">
     <object class="manifold.swing.editors.DoubleEditor">
      <void property="propertyName">
       <string>line.width</string>
      </void>
     </object>
    </void>
    <void method="add">
     <object class="manifold.swing.editors.ColorEditor">
      <void property="propertyName">
       <string>line.color</string>
      </void>
     </object>
    </void>
   </object>
  </void>
</object>
```

4.2.3 manifold.impl2D.GeometricFigure

Glyphs are represented via simple geometric shapes such as rectangles and ellipses. `Manifold.impl2D.GeometricFigure` is the base class and is extended by these specific geometric figures.

4.2.4 cachedState

A strict distinction between the application domain and the presentation layers of a software package has been maintained. Glyphs do not have any state—their actual state is defined by the corresponding objects in the application domain and mirrors what the application domain object notifies it.

Glyphs, however, cache the state information in the look-up table called `cachedState`. The reason for caching is to improve performance, especially if the domain is located across the network. The look-up table represents the glyph's attributes as a set of ⟨*property*, *value*⟩ pairs. The commonly used attributes are defined in `manifold.EventFrame`, although some may be defined locally in glyphs. The `cachedState` entries are dynamically created at runtime and their values are dynamically typed.

4.3 Property Editor Modifications

The previous sections provided a background of the design of manifold. Initial implementation included two property editors namely, *value* and *color*.

- *Value* is a JButton that allows the user to change the width of a selected glyph
- *Color* is also a JButton that provides the user an option to change the color of the selected glyph.

We felt that these two property editors were too little for the application. We decided on adding two more property editors, namely *Fill Color* and *Stroke*.

Fill Color fills the glyph with a specified color. While *stroke* changes the stroke of the outer lines of the glyph. The implementation of these two properties is described in the following two sections.

These properties were incorporated by implementing two new classes namely `FillColorEditor` and `StrokeEditor` in to the `manifold.swing.editor` package. Through the `PropertyEditorsPanel` these properties will be incorporated just as the *line color* and *width* already existent in the application. However, apart from describing these classes in `manifold.swing.editors` package, changes were made in `manifold.swing.GeometryFigure` to allow these properties affect the physical properties of the selected glyph with in the application.

The following sections will describe the implementation of these classes and the modification made to the `manifold.swing.GeometryFigure.`

4.3.1 Fill Color

Fill color provides the user an option to fill the inner area of a glyph with a selected color via a color palette. The idea was incorporated by using `java.awt.Color (27).` The `Color` class is used to encapsulate colors in the default RGB color space. Every color has an implicit alpha value of 1.0 or an explicit one provided in the constructor. The alpha value defines the transparency of a color and can be represented by a float value in the range 0.0 - 1.0 or 0 - 255. An alpha value of 1.0 or 255 means that the color is completely opaque and an alpha value of 0 or 0.0 means that the color is completely transparent. In this case we use the default alpha value for an opaque color.

The palette was incorporated by using `javax.swing.JColorChooser` (28). `JColorChooser` provides a pane of controls designed to allow a user to manipulate and select a color. It contains two parts, a tabbed pane and a preview panel. The three tabs in the tabbed pane select *chooser panels*. The *preview panel* below the tabbed pane displays the currently selected color. The `JColorChooser` API also makes it easy to bring up a dialog (modal or not). More information on how to use the JColorChooser can be found at (29). The following code stub with a dialog was incorporated in to `manifold.swing.editors.FillColorEditor`:

```
Color color_= JColorChooser.showDialog(this,"Select Fill Color", value)
```

Figure 13 shows what `JColorChooser` looks like in *Java Look & Feel*.

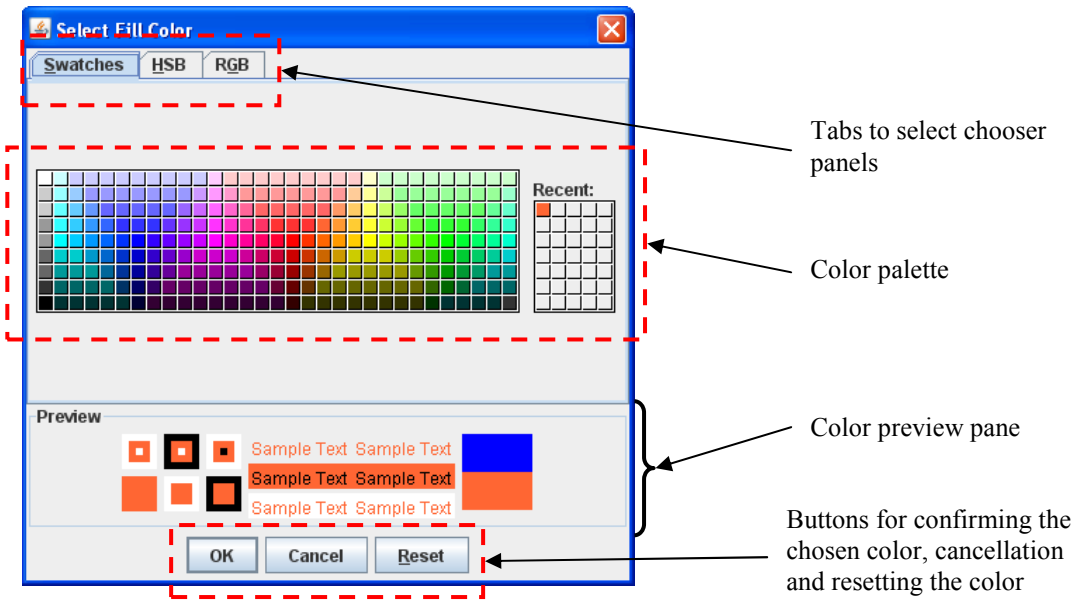

Figure 13: JColorChooser dialog box

The manifold.swing.editors.FillColorEditor implements two interfaces manifold.PropertyEditor, java.awt.event.ActionListener and extends javax.swing.JButton (30) (a push button). FillColorEditor implements ActionListener as it has to process an action event when a user clicks the JButton. Up on clicking the JButton, the ActionListener interface receives action events. An object created through the JButton is registered with a component, using the component's addActionListener method. When the action event occurs, the object's actionPerformed method is invoked. More information on how to use a JButton can be found at (31). Under the actionPerformed method I have defined the actions to be performed.

```
public void actionPerformed(ActionEvent event_) {
	Color color_ =
		JColorChooser.showDialog(this, "Select Fill Color", value);

	if (color_ != null) {
		// Make an event frame to request the application domain
		// for property change.
		Hashtable slots_ = new Hashtable();
		slots_.put(EventFrame.VERB, ControllerImpl.SET_PROPERTIES);
		slots_.put(EventFrame.NODE_ID, currentNodeId);
		slots_.put(propertyName, color_);
			this.setBackground(color_); // Change the button
	color to the current color
		propertiesViewer.getController().sendAsyncEvent(new
EventFrame(slots_));

	}
}
```

In the above code stub, the first action performed is to retrieve the color chosen by the user via the JColorChooser and store the Color value in color_. The following steps generate a manifold.EventFrame and send it to the Controller. An EventFrame contains information about the interpretations of the user's event

which has been transcribed to a form that the application can understand. In this case, the *property name* (`fill.color`) and the fill color chosen are sent to the controller via the `EventFrame`. Further details on `EventFrame` have been covered in the next chapter. Moreover the color of the `JButton` is changed to the current color chosen by

```
this.setBackground(color_);
```

The values specified in the `EventFrame` are stored in the `cachedState` *hash table* if they do not exist or are updated if they exist in it. The currently selected color has not been used by a glyph or in other words modified the properties of the glyph. It has only been chosen and is stored in the `cachesState` hash table.

There is no default fill color set to the glyph when it is created. It is the developer's responsibility to set these colors to the glyph. In order to incorporate the fill color properties to a glyph, small changes have to be made to `manifold.Impl2D.GeometricFigure`. The method `draw` renders the glyphs of the type `java.awt.Shape (32)`. Every time a glyph has to be created or modified, this method is called. In order to incorporate a property change to the glyph a change has to be made in this method.

```
Color fillColor_ = (Color) cachedState.get(EventFrame.FILL_COLOR);
if (fillColor_ != null) {
      graphics_.setColor(fillColor_);
      graphics_.fill(shape);
}
```

The above code stub is the lines of code added to `draw`. The code checks whether `cachedState` contains a fill color value. If it doesn't exist, nothing is done. However,

if it exists the color chosen by the user is set as the *fill color* to the glyph. The glyph is then re-rendered on the workspace.

Figure 14 provides few screenshots of the new *fill color* property editor.

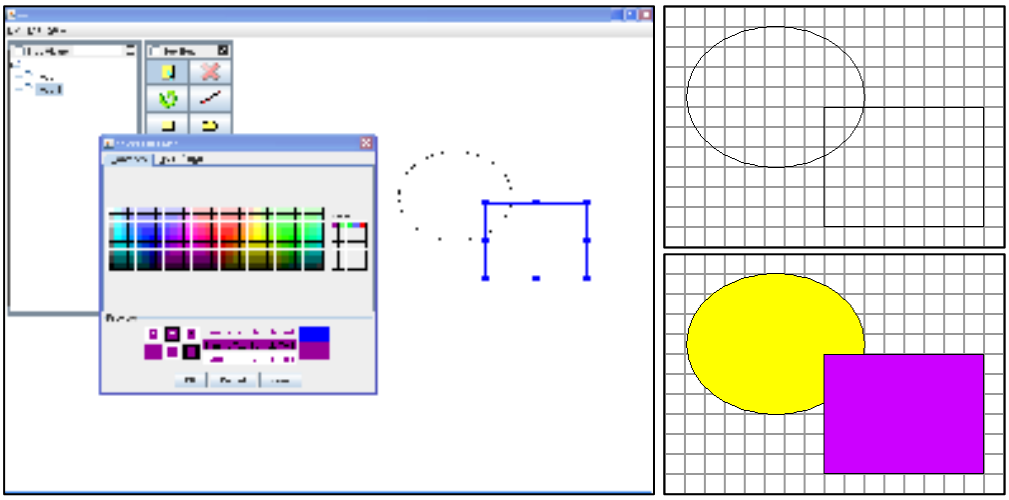

Figure 14: Screenshots of the fill color property editor. (Left) Manifold user interface with the fill color palette (Top right) Two glyphs with default properties, before the fill color has been applied (Bottom Right) Two glyphs with fill color applied.

4.3.2 Stroke

Once *fill color* was complete, we felt the glyphs were very plain and needed some decorations. We wanted to add a feature that could decorate the outer lines of the glyph. This consequently resulted in the implementation of a *stroke editor*.

Stroking a `Shape` is like tracing its outline with a marking pen of the appropriate size and shape. The area where the pen would place ink is the area enclosed by the outline `Shape`. The `BasicStroke (33)` class defines a basic set of rendering attributes for the outlines of graphics primitives, which are rendered with a `Graphics2D` object that has its Stroke attribute set to this `BasicStroke`. The rendering attributes defined by

`BasicStroke` describe the shape of the mark made by a pen drawn along the outline of a `Shape` and the decorations applied at the ends and joins of path segments of the `Shape`. These rendering attributes include:

- *width* - The pen width, measured perpendicularly to the pen trajectory.
- *end caps* - The decoration applied to the ends of unclosed sub paths and dash segments. The three different decorations are: `CAP_BUTT`, `CAP_ROUND`, and `CAP_SQUARE`.
- *line joins* - The decoration applied at the intersection of two path segments and at the intersection of the endpoints of a sub path. The three different decorations are: `JOIN_BEVEL`, `JOIN_MITER`, and `JOIN_ROUND`.
- *miter limit* - The limit to trim a line join that has a JOIN_MITER decoration. A line join is trimmed when the ratio of miter length to stroke width is greater than the miterlimit value. The miter length is the diagonal length of the miter, which is the distance between the inside corner and the outside corner of the intersection. The smaller the angle formed by two line segments, the longer the miter length and the sharper the angle of intersection. The default miterlimit value of 10.0f causes all angles less than 11 degrees to be trimmed. Trimming miters converts the decoration of the line join to bevel.
- *dash attributes* - The definition of how to make a dash pattern by alternating between opaque and transparent sections.

All attributes that specify measurements and distances controlling the shape of the returned outline are measured in the same coordinate system as the original un-stroked

Shape. When the Graphics2D object uses a Stroke object to redefine a path during the execution of the draw method, the geometry is supplied in its original form before the Graphics2D transform attribute is applied. Therefore, attributes such as the pen width are interpreted in the user space coordinate system of the Graphics2D object and are subject to the scaling and shearing effects of the user-space-to-device-space transform in that particular Graphics2D. For example, the width of a rendered shape's outline is determined not only by the width attribute of this BasicStroke, but also by the transform attribute of the Graphics2D object.

The class manifold.swing.editors.StrokeEditor defines the code necessary to implement the *stroke editor*. This class implements two interfaces manifold.PropertyEditor, java.awt.event.ActionListener and extends javax.swing.JPanel. The JPanel is used to hold two JComponents, one is a JComboBox and the other is a JTextField (34). JComboBox (35) combines a button and a drop down list allowing the user to chose a value from the drop-down list, which appears at the user's request. Here, the JComboBox contains a list of editable *strokes* that the user can chose from. JTextField is a lightweight component that allows the editing of a single line of text. Here, it will allow the user to specify the width of the stroke. Both the JComboBox and JTextField are added to the ActionListener via the addActionListener method. This is because the application has to process an action event when a user clicks the JComboBox and JTextField. As in the FillColorEditor class, up on clicking either the JComboBox or JTextField, the ActionListener interface receives action events. An object created through them is registered with a component, using the

component's `addActionListener` method. When the action event occurs, the object's `actionPerformed` method is invoked. Under the `actionPerformed` method I have defined the actions to be performed.

```
public void actionPerformed(ActionEvent event_) {
      String item_ = (String) comboBox_.getSelectedItem();
      Float value_ = Float.parseFloat(text_.getText());
      float[] dash1 = {5.0f};
      float[] dash2 ={10.0f};
      float[] dash3 ={15.0f};
      if(item_ != null){
            if(item_ == strokeType_[0]){
                  stroke_ = new BasicStroke(value_);

            }

            else if(item_ == strokeType_[1]){
                  stroke_ = new BasicStroke(value_,
                  BasicStroke.CAP_ROUND, BasicStroke.JOIN_BEVEL, 10.0f,
                  dash2, 0.0f );
            }

            else if(item_ == strokeType_[2]){
                  stroke_ = new BasicStroke(value_,
                  BasicStroke.CAP_SQUARE, BasicStroke.JOIN_ROUND,
                  10.0f, dash3, 0.0f);
            }

            else if(item_ == strokeType_[3]){
                  stroke_ = new BasicStroke(value_,
                  BasicStroke.CAP_ROUND, BasicStroke.JOIN_BEVEL, 10.0f,
                  dash1, 0.0f);
            }

            else if(item_ == strokeType_[4]){
                  stroke_ = new BasicStroke(value_,
                  BasicStroke.CAP_SQUARE, BasicStroke.JOIN_MITER,
                  10.0f, dash2, 0.0f);
            }

            else if(item_ == strokeType_[5]){
                  stroke_ = new BasicStroke(value_,
                  BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER, 10.0f,
                  dash1, 0.0f);
            }

            // Make an event frame to request the application domain
            // for property change.

            Hashtable slots_ = new Hashtable();
            slots_.put(EventFrame.VERB, ControllerImpl.SET_PROPERTIES);
```

```
            slots_.put(EventFrame.NODE_ID, currentNodeId);
            slots_.put(propertyName, stroke_);
            propertiesViewer.getController().sendAsyncEvent(new
            EventFrame(slots_));
      }

}
```

Based on the *stroke* selected by the user through the `JComboBox` is assigned to `item_`.

```
      String item_ = (String) comboBox_.getSelectedItem();
```

Based on the users input, the corresponding stroke property is selected and added to the `EventFrame`. The values are then added to the `cachedState` *hash table* through the *Controller* if they do not exist or are updated if they exist in it. During the creation of a new glyph, a default stroke made available by `Graphics2D` is applied. Similar to the implementation of *fill color,* changes to the `draw` method in `manifold.impl2D.GeometricFigure` are made in order to render the new stroke. The code stubs added were:

```
BasicStroke stroke_ = new BasicStroke();
if (cachedState.containsKey(EventFrame.LINE_STROKE)){
      stroke_= (BasicStroke) cachedState.get(EventFrame.LINE_STROKE);
}

graphics_.setStroke(stroke_);
```

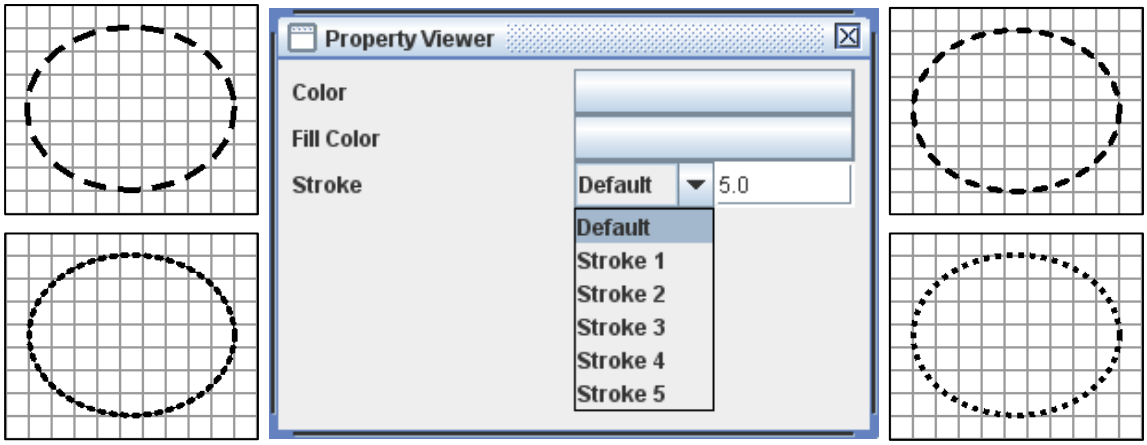

Figure 15: Property Viewer with the implementation of the Stroke editor. Surrounding the Property Viewer are screenshots of the some of the implemented strokes

The code checks for the existence of the *stroke* property in the `cachedState`. If it exists the new stroke is added to the glyph and the glyph is re-rendered other wise no action is take and the default stroke is drawn. **Figure 15** provides screenshots of the *stroke* property editor

Chapter 5

Event Frame

5.1 Tool, Manipulator, Controller

Tool encapsulates the semantics of user interaction with the application. User handling of input device(s) generates interaction events, which need to be translated to actions on the domain model. For example, the user's activity of depressing the mouse button and dragging the mouse around the workspace has different meaning, depending on the currently selected tool. Examples are rotation of a graphical figure, resizing, translation, etc. **Figure 16** displays the Tool Box that encapsulates the features provided by `Tool`.

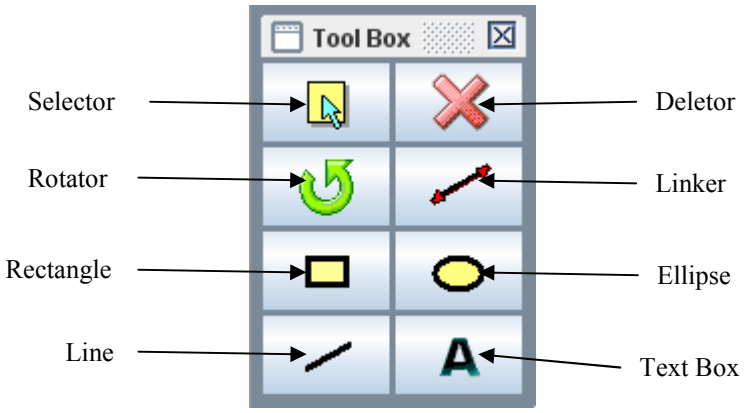

Figure 16: Tool Box encapsulates the features of Tool

In order to accomplish a user's action, a *Manipulator* is created by Tool to carry out manipulations to a glyph. In other words, Tool encapsulates *state* and Manipulator

encapsulates *behavior*. A new Manipulator is instantiated (by invoking `Tool.createManipulator()`) at the moment the user starts a new interaction cycle and disposed of at the end of the interaction cycle. An example of "interaction cycle" is: (1) user depresses a mouse button; (2) drags the mouse across the workspace; and, (3) releases the mouse button. Elaborating further: input events come from a positional device such as a mouse. When a new rectangle glyph is created, Tool initiates a new *Manipulator*. The manipulator calculates the glyphs co-ordinates in the workspace and adds them accordingly to a default glyph with preset values. Once the necessary calculations are done, the manipulator sends an *Event Frame* to the controller with the required actions and *transformations* to be performed on the domain model, in this case "*add node*".

To notify the domain model, an *Event Frame* is sent to the controller because manifold is based on a Model-View-Controller (MVC) design. The controller acts as a gate way to the domain model. Through the controller, the domain is notified about the request for addition of a new node. The domain will then perform domain specific actions and during the process (or after) will display its actions in the workspace. **Figure 17** summarizes the steps pictorially.

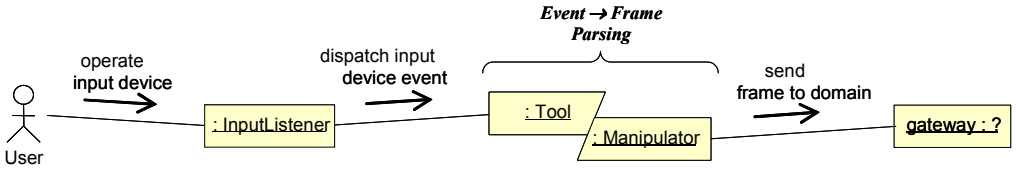

Figure 17: UML diagram summarizing typical input event interpretation in Manifold. The collaboration diagram accentuates the central role of Tool/Manipulator in this process (Marsic, I)

It can be observed that the user's actions do not manipulate the domain model directly. Instead, the users actions are translated to an *Event Frame* which is sent to the controller (gateway) requesting for a particular action to take place. `Controller` is a single object acting as a gateway between the presentation and domain modules of the system.

Since the *Event Frame* contains information about interpretations of the user's event. The controller implementation must specify a well-known list of the verbs that will be used in the event frames generated by the manipulators. The vocabulary is application-dependant and both manipulators and the application domain must know the meaning of these verbs. To be more precise, the manipulators must know how to parse the input events into the verbs (and other slots of the event frame). Application domain must know what action(s) to take in response to particular event frame. Of course, there is no need for manipulators to know neither what those actions are nor what their meaning is.

`manifold.Controller` is an interface which defines the Controller methods. It is implemented by `manifold.ControllerImpl.` In our example implementation, the following verbs are defined in `manifold.ControllerImpl:`

- public static final String ADD_NODE = "add";
- public static final String DELETE_NODE = "delete";
- public static final String SET_PROPERTIES = "setProperties";
- public static final String PROPERTY_QUERY = "propertyQuery";

Since it is the duty of the event frame to convey a users intents to the controller, it should be quick and be implemented effectively. Should be portable, and should be easily

read by an XML parser. The idea is to able to port this application on to the internet, and in web domain it is easier if such things are strings as we use an XML parser to parse it. The current implementation of the EventFrame is a hash table. The idea is to re-design it to be a string.

This chapter deals with the *Event Frame*. Its structure and modifications made to it in order to improve manifolds efficiency. The idea is to be internet viable. The current implementation of Event Frame is a Hashtable. The need is to convert it to a string.

The *viewer* uses a translation table to map the raw *event* into an action represented by an *event frame*. The *event frame* is passed on to the application domain for interpretation and execution.

5.2 Introduction to Event Frame

Event Frames are frame objects containing the information about the interpretation of the user event. Frames are normally generated by `Manipulators`, which "parse" input device events and convert them into the actions to be performed on the domain model. The recipient of a frame is usually the `Controller` object. Frames are a concept from Artificial Intelligence, introduced by Marvin Minsky of MIT (36; 37).

Though the `Manipulator` is the primary source of Event Frames, there are other classes that make use of them. The `manifold.swing.TreeViewer`, manifold.`swing.PropertiesViewer`, `manifold.swing.Viewer2DImpl` and classes with in the package `manifold.swing.editors` are the other classes that make use of Event Frames for similar purposes, to convey a message to the

`Controller`. **Figure 18** displays a UML sequence diagram describing the usage of the Event Frame in manifold.

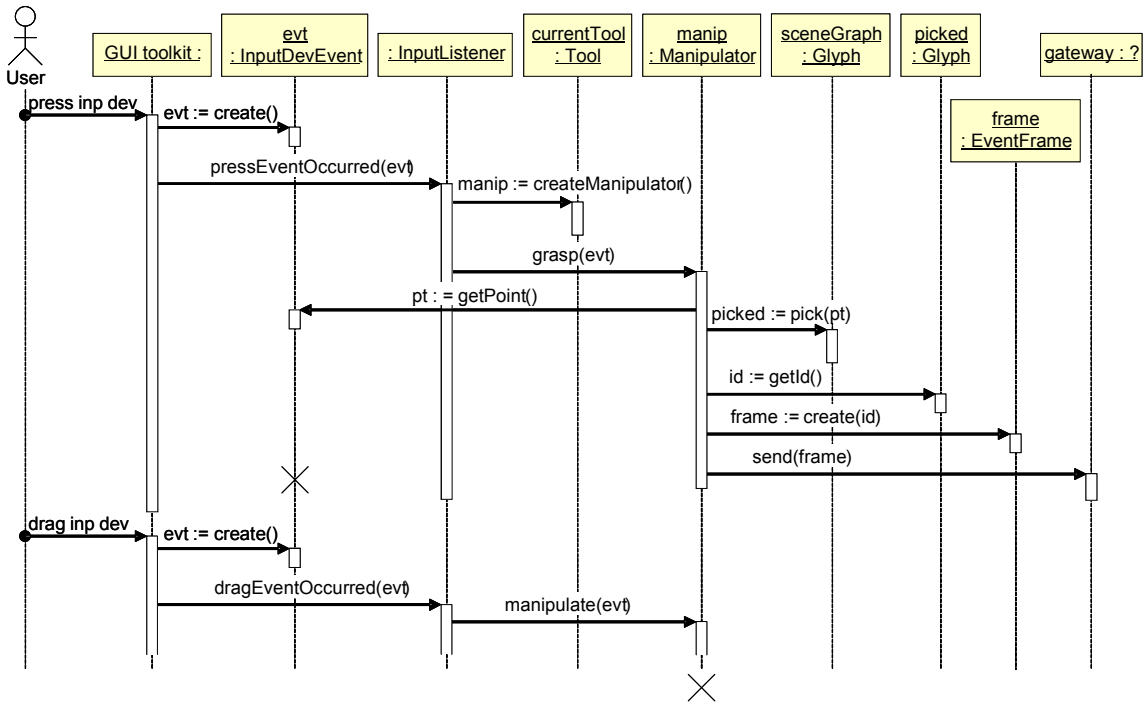

Figure 18: UML Sequence diagram showing the usage of *Event Frame* as a helper in communicating the user's intent to the domain model via `Controller` (gateway)

5.3 Design

The current implementation of manifold describes the Event Frame class in `manifold.EventFrame`. In order for the Event Frame to effectively communicate a user's actions to the Controller, the senders of an Event Frame should translate these actions to an appropriate description of the actions. These descriptions should be easily understood by the sender and the receiver because the sender needs to encode them while the receiver needs to decode them. To enable this, the Event Frame creates a table that

contains a set of *Keys* and *Values*. The keys are set of pre-defined *property slot* to which the sender has to assign *slot values*. At the Controller (receiver), based on the *slot values* of the *property slots,* the Controller requests the domain model to perform necessary actions. The *property slots* included in the Event Frame of the current implementation of manifold are:

- **`public static final String VERB = "verb",`** The verb slot of the frame. It identifies the action that the frame represents. The actions are defined in manifold.ControllerImpl as:
    - **`add`**
    - **`delete`**
    - **`setProperties`**
    - **`propertyQuery`**
- **`public static final String SOURCE = "source",`** The source slot of the frame. Also known as the active causal agent instigating the action.
- **`public static final String NODE_ID = "nodeId",`** The identifier slot which identifies the target (glyph) up on which the action is done. Note: There could be more than one target objects identified here. In such a case, the object ID's should be separated by either a coma or a white space.
- **`public static final String NODE_TYPE = "nodeType"`**, The logical type slot identifies the type of the model object. It is used to map a glyph object class when instantiating a new glyph.
- **`public static final String PARENT_ID = "parentId"`**, The parents identifier slot, identifies the "parent" object of the model object. This assumes that objects are organized in a hierarchical structure.

- **public static final String *SOURCE_URL* = "source",** The source URL slots for objects downloaded from the web.
- **public static final String *TRANSFORM* = "transform",** The transformation slot. Represents the spatial transformation that is applied to the objects visual representation
- The graphical style slots, for describing the graphical attributes of a glyph. These are:
    - **Public static final String LINE_WIDTH = "line.width"**
    - **Public static final String LINE_COLOR = "line.color"**
    - **Public static final String FILL_COLOR = "fill.color"**
    - **Public static final String LINE_STROKE = "line.stroke"**

Based on the user's actions, it is the responsibility of the sender to instantiate a new frame and assign the appropriate *slot values* to the above mentioned *property slots*. From the moment a frame is created, slot values assigned by its sender and till the frame reaches the `Controller`, it is the responsibility of the `EventFrame` to maintain the correct assignment of these slots. In order to accomplish this, the current implementation of manifold contained a `HashTable` as the data structure to hold these *key* and *value* pairs. Through the `EventFrame's` contructor, this `HashTable` is set.

```
private Hashtable slots = null;

public EventFrame(Hashtable slots_) {
      this.slots = slots_;
}
```

In order to describe a user's action, the sender of a frame needs to create a `HashTable` containing slot values of the necessary property slots while not describing the rest. It is not necessary to fill all the property slots with a value as has been described in the code stub below. This `HashTable` is then assigned to the `EventFrame` via its `contructor`. The following code stub describes the user's action of assigning a fill color to a selected glyph:

```
Hashtable slots_ = new Hashtable();
slots_.put(EventFrame.VERB, ControllerImpl.SET_PROPERTIES);
slots_.put(EventFrame.NODE_ID, currentNodeId);
slots_.put(propertyName, color_);
propertiesViewer.getController().sendAsyncEvent(new EventFrame(slots_))
```

The last line of code is the sender (in this case, `manifold.swing.editors.FillColorEditor`) sending the event frame to the `Controller` after obtaining the current `Controller` instance.

`manifold.EventFrame` describes methods that the `Controller` can use to obtain the slots. The current methods described in `EventFrame` are:

- `containsSlot`, Informs whether or not there is a slot in this frame with the specified name. Method returns `true` if such slot exists, `false` otherwise.
- `getSlots`, Accessor for retrieving all the slots of the frame at once. The method returns a HashTable containing all the slots.
- `getSlotValue`, Accessor for retrieving individual slots of the frame. The method returns the value of the slot, or `null` if this frame does not have such slot.

5.4 Issue

The design of `manifold.EventFrame` is very simple and so is its purpose. Any object with in the application that has a need to contact the Controller to convey a user event to the domain model uses `EventFrame`. The manifold interface and its domain model also being fairly simple in its current implementation means that the number of `EventFrame's` sent to the Controller are fewer and so are the number of different property slots and verbs.

The main idea behind creating manifold is to provide a generic UI which can be used over any application (domain model). As we know very well, different applications have different sets of inputs and so the UI should be able to translate a users input on manifold's workspace to that of the domain model. Moreover, manifold still being in its early stages has a lot more features that could be added to it which will be needed by other domain models. To solve these issues, the MVC design and the `EventFrame` were incorporated. Where, the `Controller` acts as a sole liaison between the manifold UI and domain model. And the `EventFrame` conveys user's intentions (event) in a standardized format to the `Controller` to be acted up on the domain model.

The `EventFrame` being the only "medium" to communicate to the `Controller` makes it pivotal in communication between the UI and the domain model. We envision manifold to grow in to a complex application that would cater to various complex domain models on the internet. In such a situation, the number of messages sent between the `manipulator` and `Controller` would be enormous. Also, the number of property slots that will be needed would increase as they require defining every event in the most appropriate way.

Given these issues, we felt that the current implementation of `manifold.EventFrame` as a `HashTable` was inappropriate. Under a small workload in its current implementation its effects on performance may not be of concern, but as the UI becomes more complex and support more intricate domain models on the web, the `Hashtable` could be a big hindrance. Moreover, choosing an effective hash function for a specific application is more an art than a science. Although operations on a hash table take constant time on average, the cost of a good hash function can be significantly higher than the inner loop of the lookup algorithm for a sequential list or search tree.

Over the internet, XML is gaining popularity as a way to share and transport data. One of the most time-consuming challenges for developers is exchanging data between incompatible systems. XML greatly reduces this complexity, since the data can be read by different incompatible applications. Owing to its wide popularity and adoption in various applications we have considered adopting it in manifold. The current implementation of manifold uses XML at a modest level. As the application moves towards a web domain XML will be adopted extensively.

As XML gets adopted more widely, efficient parsing of XML documents is more and more critical. It is very important to have an efficient way to parse XML data, especially in applications that are intended to handle large volumes. Improper parsing can result in excessive memory usage and processing times that can hurt scalability.

For parsing in Java, several types of XML parsers are available (e.g. DOM, SAX, StAX). An XML parser takes as input a raw serialized string and performs certain operations on it. This being the case, we felt that in a web domain a `HashTable` would

again not be very effective. According to (38), XML syntax is redundant to binary representations of similar data, especially with tabular data. We felt that instead of a using a `Hashtable` to represent the `EventFrame` we should use a *comma* separated `String` that would make it easier for the XML parser to parse data.

Given the above mentioned issues with the `Hashtable` we decided to replace it with a *comma* separated `String`. This chapter describes my work on this new design of `manifold.EventFrame` and the subsequent changes I made to the application for it to work.

5.5 New EventFrame Design

In the current implementation of `manifold.EventFrame`, a `java.util.Hashtable (39)` holds the *property - value slot* in a *(key, value)* respectively. We propose a *comma* separated `java.lang.String (40)` to hold those values instead. **Figure 19** summarizes the idea.

KEY	VALUE
Property 1	*Value 1*
Property 2	*Value 2*
Property 3	*Value 3*
Property 4	*Value 4*
Property 5	*Value 5*

(a)

(Property 1, Value 1, Property 2, Value 2, Property 3, Value 3, Property 4, Value 4, Property 5, Value 5)

(b)

Figure 19: A pictorial representation of (a) `HashTable` **that is used in the current implementation of** `EventFrame` **(b) the new proposed** `String` **implementation**

Figure 19-(b) describes the new idea, where the *property - value slots* are placed adjacent to each other separated by a *comma*. Moreover, the property and values in the slots are also separated by commas. While this may seem a trivial task of replacing a `HashTable` with a `String`, it wasn't as simple it seemed. Primarily because of the applications extensive dependence on `manifold.EventFrame`. Manifold relies on `EventFrame` in sending varied information to the `Controller.` This information may also be a `Java.lang.Object (41)` and not just a `String` as described in earlier sections. The current implementation being a `java.util.HashTable` had no issues in holding a `java.lang.Object` as one of its *Value*'s. However, in the proposed new idea a `java.lang.Object` is unacceptable with in `java.lang.String`! The second issue that I faced was regarding the vast code changes that I had to make in order to make the various Classes of manifold to use the new `EventFrame`. The issues faced, and the solution to these problems has been explained in further sections.

5.6 Re-Engineering: New EventFrame Format

With out changing the basic structure of the `EventFrame`, I replaced the `HashTable` with a `java.lang.String`. The resulting changes to the constructor were,

```
private String slots = null;
public EventFrame(String slots_) {
      this.slots = slots_;
}
```

The `getSlot` method in EventFrame which returns the frame was implemented by replacing the return value to a `String`.

```
public String getSlots() {
	return slots;
}
```

The `containsSlot` method checks for the existence of a slot with a specified name in the frame. In the old design, to check for existence of a slot a `HashTable` method `containsKey` was used. It returns a `Boolean true` if it exists else `false`. `Java.lang.Strings` does not provide a method which searches for the existence of a sub-string and returns a `Boolean` value. To achieve this a few extra lines of code had to be written. The results were achieved by using the `indexOf` method in `java.lang.String`. `indexOf` method returns the index of the first occurrence of the specified substring. If the substring argument occurs with in this `String`, the index of the first character of the first such substring is returned; if it does not occur as a substring, `-1` is returned. Since a particular slot is specified only once in a frame, the `indexOf` method is appropriate (40). The `containsSlot` method was implemented using the `indexOf` method on the `String` and checking for a "-1" return value. If -1 was returned a `Boolean false` was returned by the `containsSlot` method else `true`.

```
public boolean containsSlot(String slotName_) {
	boolean find = false;
	if(slots.indexOf(slotName_)!= -1){
		find = true;
	}
	return find;
}
```

The third method, `getSlotValue` returns individual slots of the frame. Under the old design this was easily implemented by using the `get` method of `HashTable`. In the new design, every slot is *comma* separated and all the slots are placed adjacent to each other also separated by a *comma* (as shown in **Figure 19-(b)**). To search for a slot value, the best way would be to break the `String` in to tokens and search for the slot name. Once obtained the next token will be the correct slot value. Tokens are small chunks of words formed from a `String` based on a simple rule. In this case, the comma defines the rule and breaks the `String` in to tokens. This idea was achieved by using `java.util.StringTokenizer`. The `StringTokenizer` class allows an application to break a `String` into tokens. The delimiter (rule) was set via the contructor of the class. A code stub of the method has been provided below,

```
public String getSlotValue(String slotName_) {
	String slotValue_ = null;
	st_ = new StringTokenizer(slots,",");

	// checks whether the slot with slotName_ exists
	if(containsSlot(slotName_)){
		while(st_.hasMoreTokens()){
			slotValue_ = st_.nextToken();

		// Checks for slotName_. When found return the next token
			if(slotValue_.equals(slotName_)){
				slotValue_ = st_.nextToken();
				break;
			}
		}
	}

	return slotValue_;
}
```

Till now, changes were made only to the existent methods of `EventFrame`. While these methods were sufficient for `EventFrame` under the old design, they weren't for the new design. Manifold uses `Hashtables` at several locations. For example, the `cachedState` described in earlier sections is a `Hashtable` which holds properties of a glyph. And the Controller has a `Hashtable` called *model* which is a Hashtable of hashtables that simulates the domain model. In the earlier implementation, the `EventFrame` being a hash table resulted in several direct assignments of the `EventFrame` to these and other `hashtables`. Such as,

1. `model.put(id_, frame_.getSlots().clone());`
2. `Hashtable newProps_ = frame_.getSlots()`

I order to make these assignments viable with out greatly changing the code of manifold itself; I implemented a new method called `getHashSlots`. `getHashSlots` returns a HashTable representation of the String frame. This was implemented as follows,

```
public Hashtable getHashSlots(){
      Hashtable hashtable_= new Hashtable();
      if(slots != null){
            st_ = new StringTokenizer(slots,",");
            while(st_.hasMoreTokens()){
                  String key_ = st_.nextToken();
                  String value_ = st_.nextToken();
                  hashtable_.put(key_, value_);
            }
      }
      return hashtable_;
}
```

In a `Hashtable`, the `put` method maps a specified `key` to a specified `value`. If the `key` already exists then the previous `value` of the key is replaced with the new

value and the old value is returned. A `remove` method on the other hand removes the key-value pair from the hash table and returns the old value. These two methods were occasionally used in the `Controller`. In order to cater to these requirements I created two new methods namely `remove` and `replace` that would provide the same effects in the `String` implementation of `EventFrame`. As in a `HashTable`, the `remove` method in the new design removes a specified slot from the `String` and the `replace` method replaces a specified slot with a new slot value. Code stubs of the `remove` and `replace` methods have been provided below,

```
public String remove(String slotName_){
	String key_;
	String value_ = null;
	String tempSlot_ = new String();
	st_ = new StringTokenizer(slots,",");
	if(containsSlot(slotName_)){
		while(st_.hasMoreTokens()){
			key_ = st_.nextToken();
			value_ = st_.nextToken();
			if(key_.equals(slotName_)){
				continue;
			}

			else{
			tempSlot_ = (tempSlot_ + key_ + "," + value_ + ",");
			}
		}
	}
	//Ensures that the last "," is removed.
	slots = tempSlot_.substring(0, (tempSlot_.length()-1));;
	return value_;
}
```

```
public String replace(Object slotName_, Object newValue_){
	String oldValue_ = null;
	if(containsSlot((String)slotName_)){
		oldValue_ = remove((String)slotName_);
		slots = (slots + "," + (String)slotName_ + "," +
		(String)newValue_);
	}
	return oldValue_;
}
```

5.7 Modifications to Manifold

Earlier sections of this chapter had discussed `EventFrame's` role in manifold. It required a sender to create a new frame containing all the relevant event information and a receiver that would receive this frame and translate the information to the domain model. In manifold the receiver is the `Controller` while the sender is typically the Manipulator. However, there are instances from the `PropertiesViewer`, `TreeViewer` and `manifold.swing.editors` package that use `EventFrame`. These Classes till now depended on the old `EventFrame` format to communicate. Which means the application by itself was designed to recognize the `EventFrame` as a `Hashtable`. Following the changes to the `EventFrame`, the manifold application wouldn't cater to the new design of `EventFrame` because of the new `String` format. For instance, in a typical case of creating an `EventFrame` the sender would first create a `HashTable` containing all the necessary slots. It would then assign these slots to a new instance of `EventFrame` and send the frame to the `Controller`, as shown in the following code stub

```
Hashtable slots_ = new Hashtable();
slots_.put(EventFrame.VERB, ControllerImpl.SET_PROPERTIES);
slots_.put(EventFrame.NODE_ID, currentNodeId);
slots_.put(propertyName, color_);
propertiesViewer.getController().sendAsyncEvent(new EventFrame(slots_))
```

Under the new format creating a frame with the same information is a trivial task, which was obtained by

```
String slots_ = new String();
slots_ = (EventFrame.VERB + "," + ControllerImpl.SET_PROPERTIES + ","
        + EventFrame.NODE_ID + "," + currentNodeId + ","
        + propertyName + "," + color_.getRGB() );
propertiesViewer.getController().sendAsyncEvent(new EventFrame(slots_))
```

Creating a new frame seems easy as long as the contents of the slots are also `Strings`. However, this wasn't the case every time. The above code stubs are an example of this situation. Under the slot *propertyName* an Object of `java.awt.Color` was being assigned. While this was fine with a `Hashtable`, this is totally unacceptable with a `String`. A `String` can not hold any other `Object` other than a `String`. If the `Object` assigned to it isn't a `String`, it will have to be converted to a `String` before being assigned. Such assignments of `Objects` to the `EventFrame` were common through out the application and had to be dealt with in order to enable the application to function.

Once an Object has been converted to a `String`, it is difficult to obtain the same instance of the `Object` back. One of the options was to use the Java Reflections API but even they wouldn't solve the problem entirely. Another idea was to re-consider the format of `EventFrame` to hold both `String` slots and `Hashtable` slots. Where the `Hashtable's` would contain the non `String` `Objects` and the `String` slots would contain all the `Strings`.

```
String slots_ = new String();
Hashtable hashSlots_ = new Hashtable();
slots_ = (EventFrame.VERB + "," + ControllerImpl.SET_PROPERTIES + ","
        + EventFrame.NODE_ID + "," + currentNodeId);
hashSlots_.put(propertyName, color_);
```

```
propertiesViewer.getController().sendAsyncEvent(
new EventFrame(slots_, hashSlots_));
```

But this didn't suit the purpose either as it wouldn't be any different from the original `EventFrame` format and hence XML "unfriendly". It was subsequently realized that in order to incorporate the new `EventFrame` format a bold step of changing parts of Manifold had to be taken.

The following sections describe the changes made to various classes of manifold.

5.7.1 manifold.ControllerImpl

`Controller` is an interface implemented via the `manifold.ControllerImpl`. Two methods mainly deal with `EventFrame` namely, `sendAsyncEvent` and `sendSyncEvent`. `sendAsyncEvent` asynchronously sends a given event frame to the domain model. While the `sendSyncEvent` synchronously sends a given event to the domain model. Since the current implementation of `Manifold.ControllerImpl` does not have a domain model attached, the event frames are stored in a `Hashtable` called model. Since the previous implementation of `EventFrame` was a `Hashtable`, the `EventFrame` was stored in it as a `Hashtable`.

```
model.put(id_, frame_.getSlots().clone());
```

Following the changes, the model was designed to an `EventFrame` as an `Object`.

```
model.put(id_, frame_);
```

5.7.2 manifold.impl2D.tools.*

`manifold.impl2D.tools` is a package which holds the Classes describing the implementations of the various tools of manifold. These are the manipulators and are the primary source of `EventFrame's`. Changes were made regarding the creation of an `EventFrame`. The changes made in this context were regarding the assignment of slots to `Strings` instead of a `Hashtable`.

The new format was,

```
String slots_ = (EventFrame.VERB+","+ControllerImpl.SET_PROPERTIES+ ","
            + EventFrame.NODE_ID + "," + currentGlyph.getId() + ","
            + EventFrame.TRANSFORM + "," + stringMatrix_);

viewer.getController().sendAsyncEvent(
                        new EventFrame(slots_));
```

Instead of,

```
Hashtable slots_ = new Hashtable();
slots_.put(EventFrame.VERB, ControllerImpl.SET_PROPERTIES);
slots_.put(EventFrame.SOURCE, this);
slots_.put(EventFrame.NODE_ID, currentGlyph.getId());
slots_.put(EventFrame.TRANSFORM, flatmatrix_);

viewer.getController().sendAsyncEvent(
                        new EventFrame(slots_));
```

A transformation matrix is a double array holding the transformation of the glyph in the local co-ordinates. The manipulator being the implementer of various tools had to deal extensively with the transformation matrix. This being an `Array Object`, it could

not be sent via a `String` due to reasons explained earlier. To over come this issue, I incorporated two methods with in the `EventFrame` that would convert a double to a semi-colon separated `String`

```
public String editToFlatString(double[] flatMatrix_){
String flatString_ = "";
      for(int i_=0; i_<flatMatrix_.length;i_++){
      flatString_= (flatString_ + String.valueOf(flatMatrix_[i_])+";");
      }

return flatString_;
}
```

And a method to get back the double from a `String`

```
public double[] editToFlatmatrix(String stringMatrix_){
      st_ = new StringTokenizer(stringMatrix_, ";");
      double[] flatMatrix_ = new double[st_.countTokens()];
      int i_ = 0;
      while(st_.hasMoreTokens()){
            flatMatrix_[i_] = Double.parseDouble(st_.nextToken());
            i_++;
      }
      return flatMatrix_;
}
```

Each time a transformation matrix had to be sent via an `EventFrame`, it was converted to a semi-colon separated `string` before appending it to the `String`. And following it reaching its intended destination it was reconverted back to a double array.

5.7.3 manifold.swing.editors.*

This package holds the Classes describing the property editors. These Classes incorporated the `EventFrame` to send property change events to the `Controller`.

The properties are generally Java `Objects` of type `Color`, `Stroke` etc. While under the old implementation of `EventFrame`, these `Objects` were directly added to the `Hashtable` this would be inappropriate with the new implementation. So instead of sending a `Color Object` I sent its RBG and instead of sending the `Stroke` as an `Object` I converted it to a semi-colon separated String and appended them to the `EventFrame`. Following the changes, changes also had to be made to the classes that set these properties to the appropriate selected Glyph namely `manifold.impl2D.GeometricFigure`.

In `manifold.impl2D.GeometricFigure`, code stubs before and after the changes have been provided below

For Fill Color, before the `EventFrame` changes

```
Color fillColor_ = (Color) cachedState.get(EventFrame.FILL_COLOR);
if (fillColor_ != null) {
      graphics_.setColor(fillColor_);
      graphics_.fill(shape);
}
```

Changed to

```
if (cachedState.get(EventFrame.FILL_COLOR) != null) {
      Color fillColor_ = new Color(
      Integer.parseInt(cachedState.get(EventFrame.FILL_COLOR).toString(
      )));
      graphics_.setColor(fillColor_);
      graphics_.fill(shape);
}
```

For `BasicStroke`, before the `EventFrame` changes

```
if (cachedState.containsKey(EventFrame.LINE_STROKE)){
      stroke_= (BasicStroke) cachedState.get(EventFrame.LINE_STROKE);
}
graphics_.setStroke(stroke_)
```

Changed to,

```
if (cachedState.containsKey(EventFrame.LINE_STROKE)){
      StringTokenizer st_ = new
      StringTokenizer(cachedState.get(EventFrame.LINE_STROKE).toString(
      ), ";");
      float width = Float.parseFloat(st_.nextToken());
      int cap = Integer.parseInt(st_.nextToken());
      int join = Integer.parseInt(st_.nextToken());
      float miterLimit = Float.parseFloat(st_.nextToken());
      float[] dash = {Float.parseFloat(st_.nextToken())};
      float dashPhase = Float.parseFloat(st_.nextToken());
      BasicStroke stroke_ = new BasicStroke(width, cap, join,
      miterLimit, dash, dashPhase);
      graphics_.setStroke(stroke_);
}
```

5.8 Performance

Following the code changes, a performance test was performed between the old manifold UI and the new manifold UI with the modified `EventFrame` format. These tests were performed to test the effectiveness in terms of time for the new `EventFrame` to send the users information to the controller.

These experiments describe the time take to complete one full interaction cycle. A description of the interaction cycle and the results has been provided in the following sections.

Tests were performed on my laptop with the following configurations:

Processor: Mobile Intel (R) Pentium (R) 4

Speed: 3.06 GHz
RAM: 512 MB
Operating System: Microsoft Windows XP Service Pack 2 (Version 2002)
Java: Java Development Kit (JDK) 6
Platform: Eclipse SDK, Version 3.0.4

5.8.1 Application Loading Time

The time in *milliseconds* taken by both the applications to load was tested. The results are based on 5 trials. Results have been displayed in the graph below.

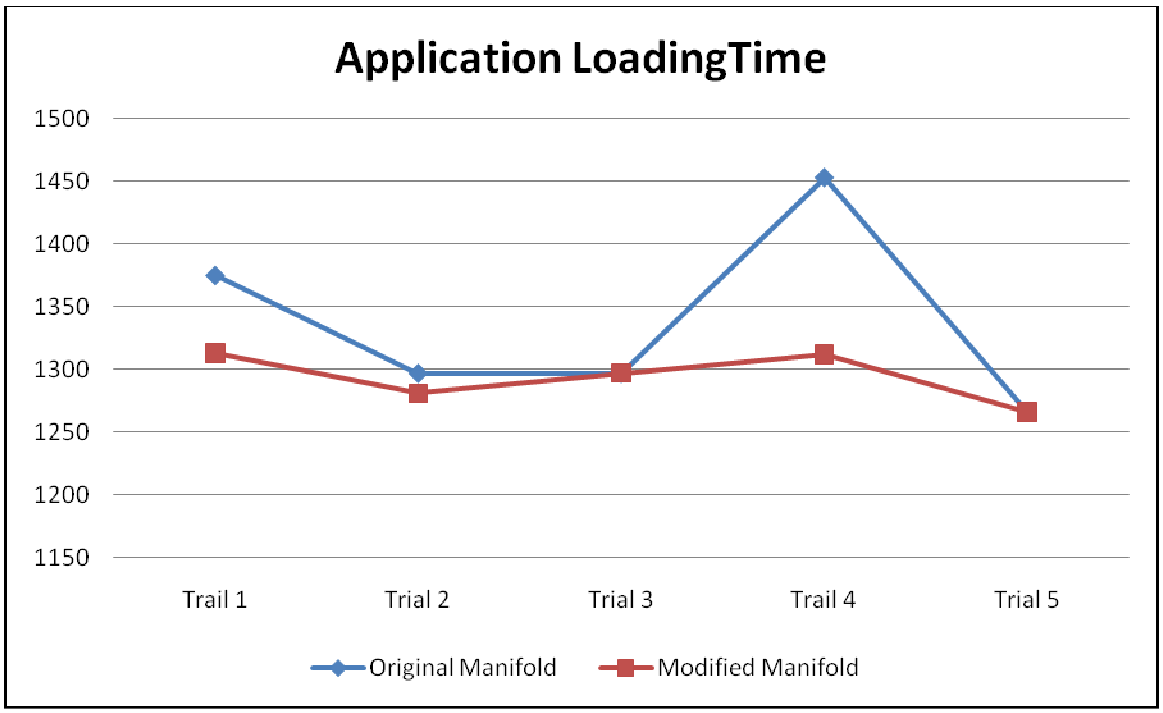

Figure 20: Displays results of the application startup time of the Original manifold and Modified manifold with the new `EventFrame`. Y-Axis displays time in milliseconds (ms) and X-Axis are the trials.

5.8.2 EventFrame Performance: Selector Manipulator

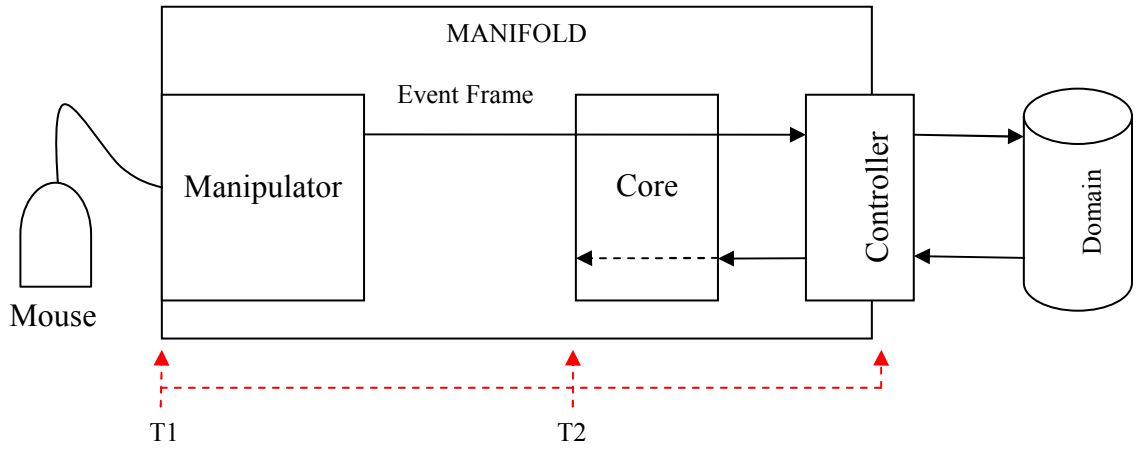

Figure 21: Displays the points between which the measurements were made in order to calculate the time take by the selector manipulator to make and display the selection.

The `Selector` *manipulator* is responsible for selecting a glyph. The results shown below describe the time taken for the applications to "consider" a glyph selected once a user has selected it. This happens when the application acknowledges the change. The figure above displays the points between which the test was conducted.

Time has been measured from the instance an event is generated by the user clicking on the glyph (sends an event) till the application acknowledges the users action and displays the result on the screen. During the process `EventFrame's` are sent to the `Controller` by the `Selector` *Manipulator*.

In order to record the results, 5 trials were conducted. The average time taken by the system to display a selected glyph was calculated by using the formula:

Time Interval = T2 - T1

Avg. Time = $\frac{(Trial\ 1+Trial\ 2+Trial\ 3+Trial\ 4+Trial\ 5)}{5}$

Avg. Time for Original Manifold = 127.20 ms

Avg. Time for Modified Manifold = 61.80 ms

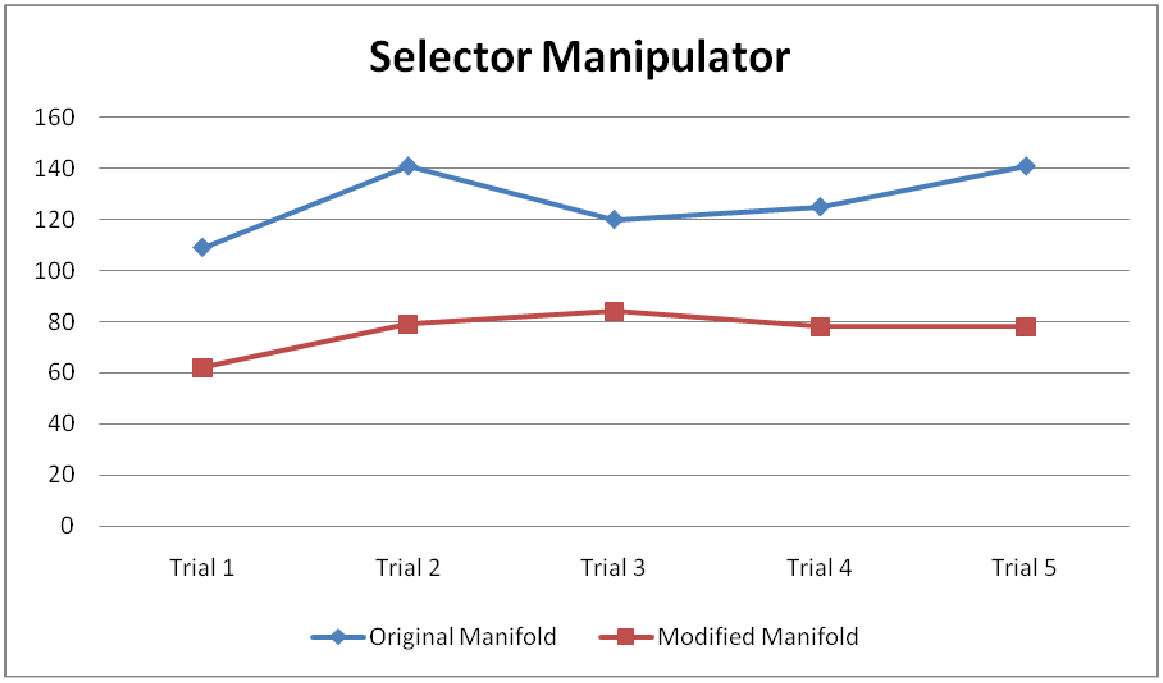

Figure 22: Displays results of the performance of EventFrame with the *Selector Manipulator* of the Original manifold and Modified manifold with the new `EventFrame`. Y-Axis displays time in milliseconds (ms) and X-Axis are the trials.

5.8.3 EventFrame Performance: Fill Color Editor

Fill Color fills the interior of a glyph with a specified `Color`. The results shown below describe the time taken by the application to fill a glyph with a specified `Color`. The figure above displays the method used to calculate the performance of the fill color editor.

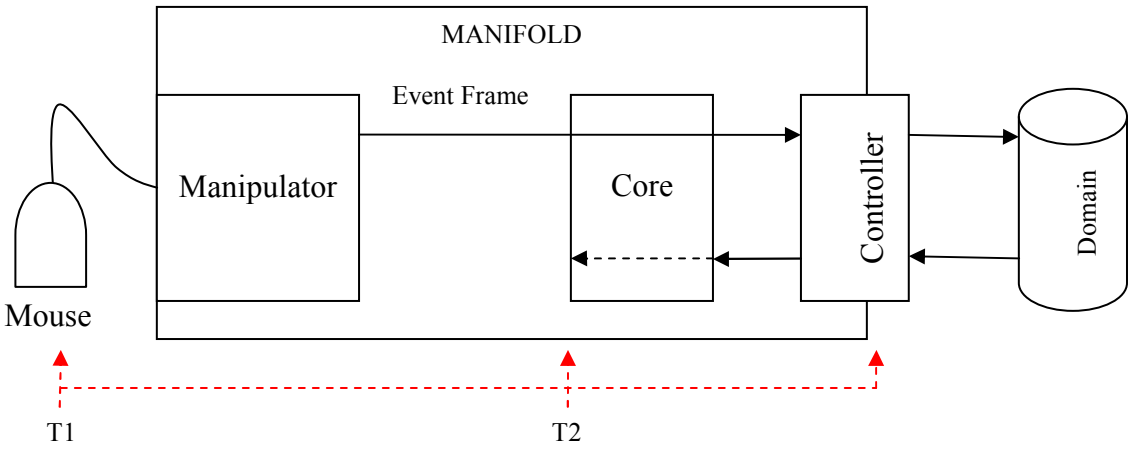

Figure 23: Displays the points between which the measurements were made in order to calculate the time take by the Fill Color editor to add a fill color to the selected glyph.

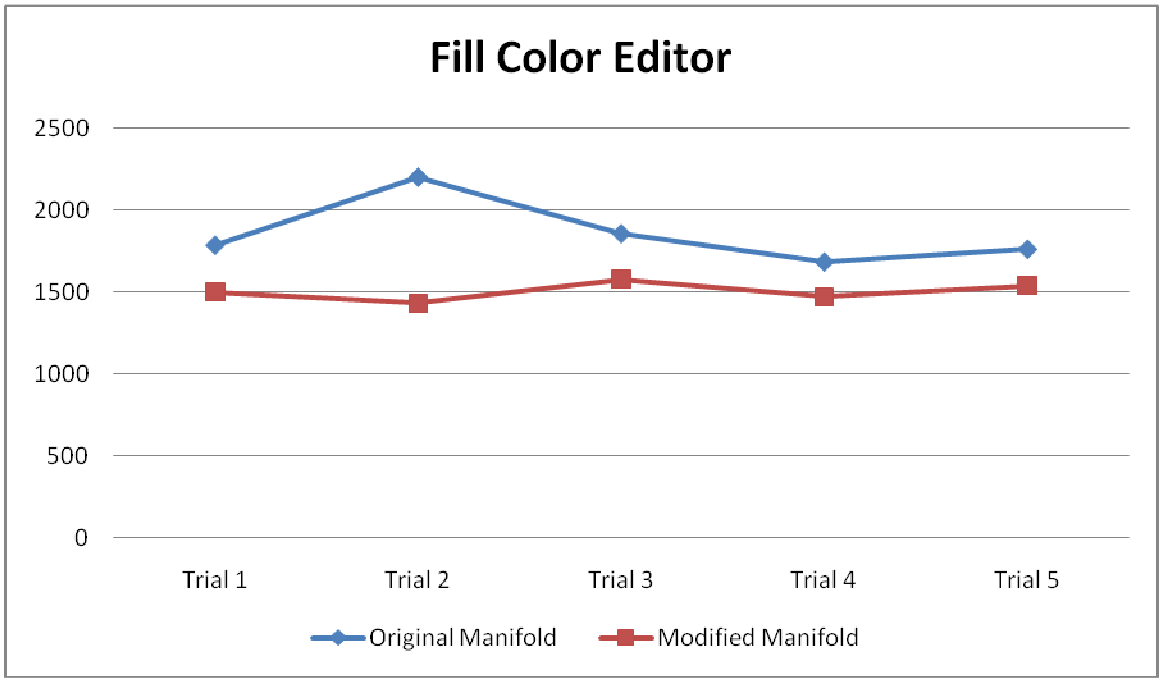

Figure 24: Displays results of the performance of EventFrame with the *Fill Color Editor* of the Original manifold and Modified manifold with the new `EventFrame`. Y-Axis displays time in milliseconds (ms) and X-Axis are the trials.

In order to record the results, 5 trials were conducted. The average time taken by the system to display a selected glyph was calculated by using the formula:

Time Interval = T2 - T1

Avg. Time = $\frac{(Trial\ 1+Trial\ 2+Trial\ 3+Trial\ 4+Trial\ 5)}{5}$

Avg. Time for Original Manifold = 1861.20 ms

Avg. Time for Modified Manifold = 1503.80 ms

5.8.4 EventFrame Performance: Creator Manipulator

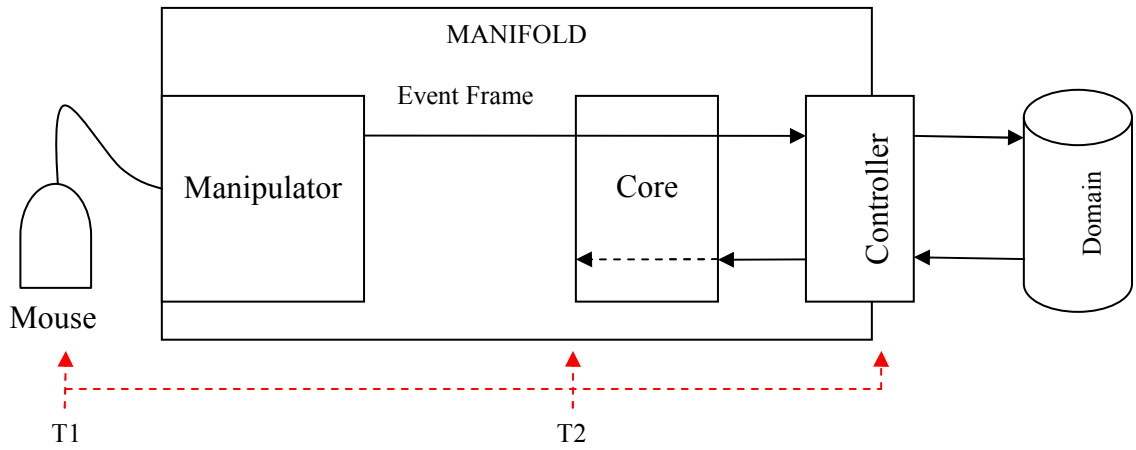

Figure 25: Displays the points between which the measurements were made in order to calculate the time taken to create a glyph and display it on the screen.

Every time a glyph is created an `EventFrame` is generated. These tests display the time taken to send 10 `EventFrame's` between the *Creator Manipulator* and The `Controller`. The figure above displays the technique used to measure the time taken to create a glyph.

Time was measured between the moments the user drags the mouse over the workspace by clicking on it till he releases it. In order to maintain consistency in this drag

and release process, I used only one click which results in a default glyph. The interval of time between 10 such glyphs was measured.

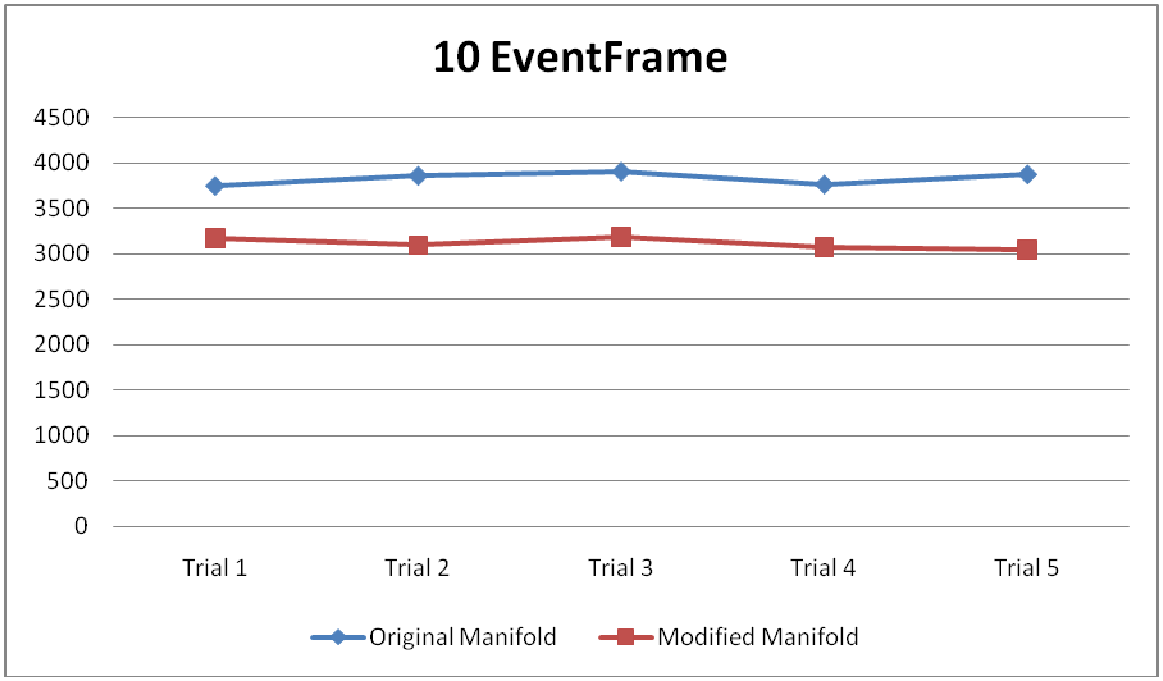

Figure 26: Displays results of the performance of 10 `EventFrame`'s of the Original manifold and Modified manifold with the new `EventFrame`. Y-Axis displays time in milliseconds (ms) and X-Axis are the trials.

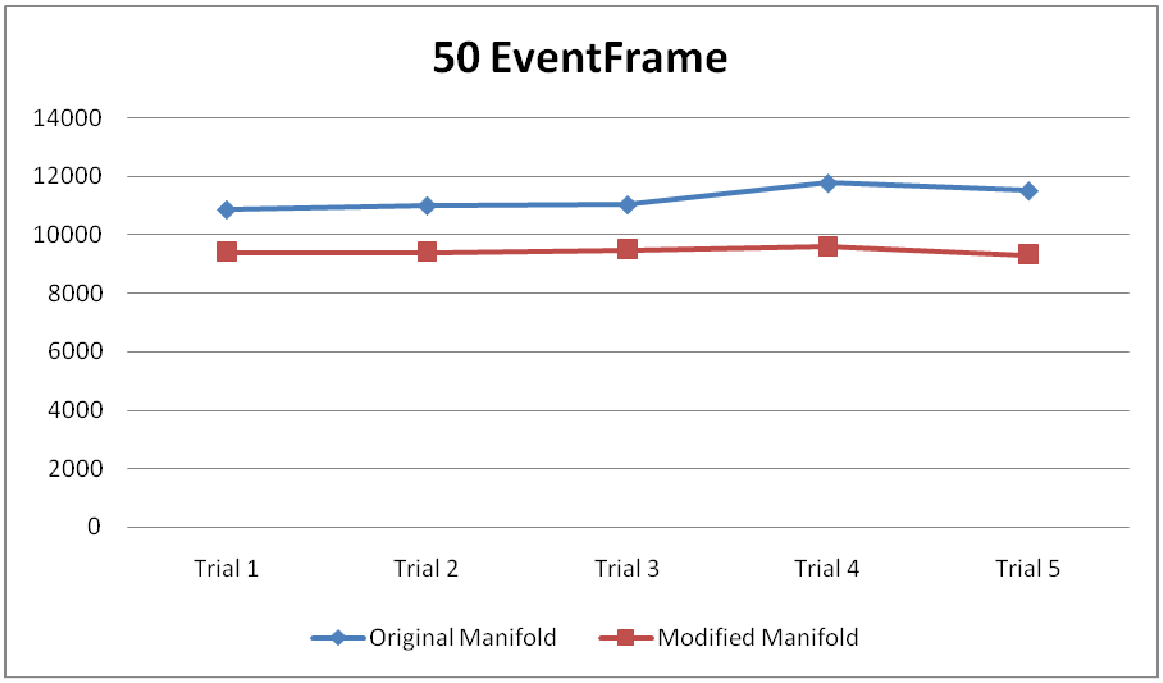

Figure 27: Displays results of the performance of 50 `EventFrame's` of the Original manifold and Modified manifold with the new `EventFrame`. Y-Axis displays time in milliseconds (ms) and X-Axis are the trials.

The above graph displays the time taken to send 50 `EventFrame's` between the *Creator Manipulator* and The `Controller`.

Time was measured between the moments the user drags the mouse over the workspace by clicking on it till he releases it. In order to maintain consistency in this drag and release process, I used only one click which results in a default glyph. The interval of time between 50 such glyphs was measured.

In order to record the results, 5 trials were conducted for a set of 10 Event Frames generated and 50 Event Frames generated. The average time taken by the system to display a selected glyph was calculated by using the formula:

Time Interval = T2 - T1

$$\text{Avg. Time} = \frac{\frac{Trial\ 1+Trial\ 2+Trial\ 3+Trial\ 4+Trial\ 5}{50}+\frac{Trial'\ 1+Trial'\ 2+Trial'\ 3+Trial'\ 4+Trial'\ 5}{250}}{5}$$

Avg. Time for Original Manifold = 303.95 ms

Avg. Time for Modified Manifold = 250.35 ms

5.8.5 EventFrame Performance: Drag Drop Feature

These tests were performed by dragging the glyphs *handle* along the workspace hence re-sizing the glyph. A *handle* are the anchor points on the glyph that are used to re-size the glyph.

Each time the handle is moved along the work space, a set of Event Frames are generated by the `Selector` Manipulator which sends then to the `Controller`. The measurements here were taken by measuring the time taken by the `Selector` to send 1000, 1500 and 2000 Event Frames to the `Controller`.

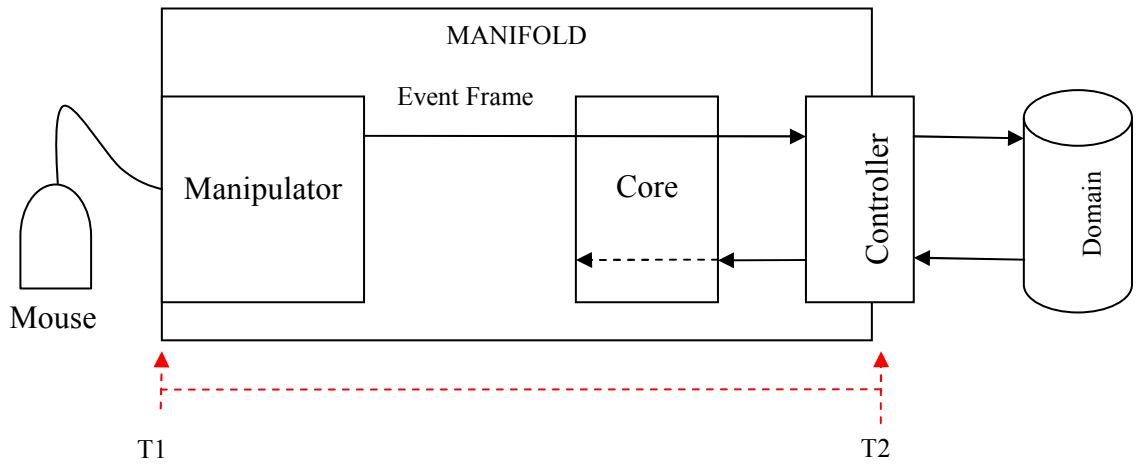

Figure 28: Displays the points between which the measurements were made in order to calculate the time taken by the Event Frame to be transported between the Manipulator and Controller.

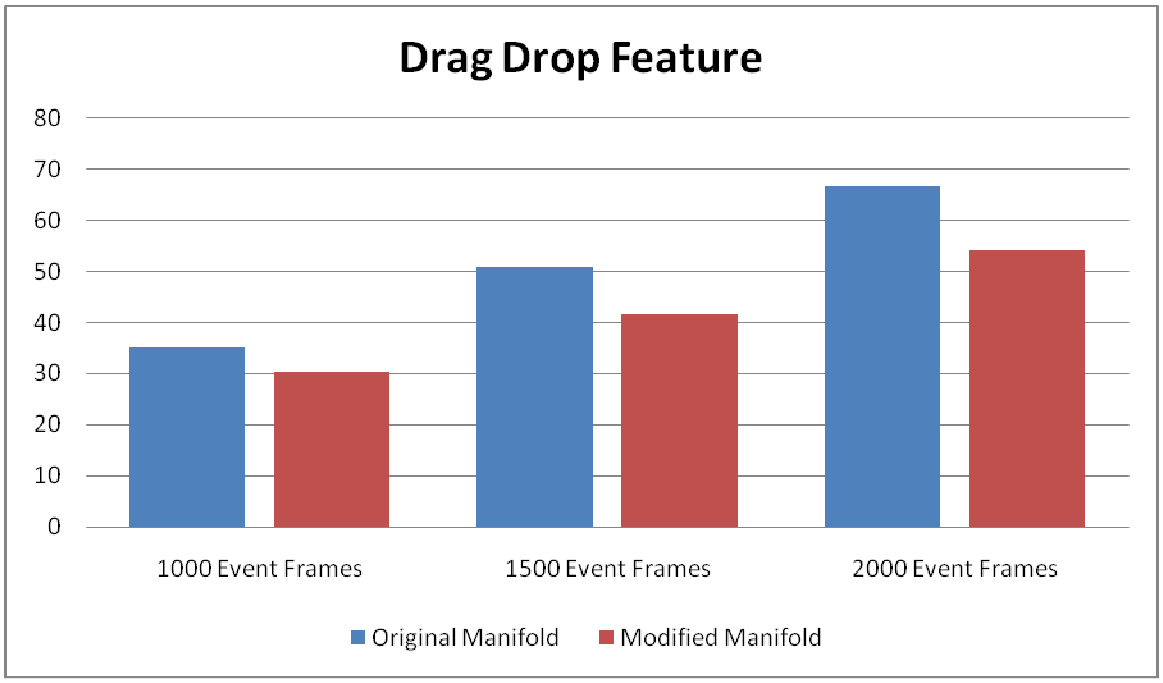

Figure 29: Displays results of the performance of `EventFrame's` of the Original manifold and Modified manifold using the `Selector` tool. Y-Axis displays time in milliseconds (ms) and X-Axis are the number of `EventFrames`.

In order to record the results, trials were conducted for a set of 1000 Event Frames, 1500 Event Frames and 2000 Event Frames. The average time taken by the system to send an Event Frame across the system is:

Time Interval = T2 - T1

Avg. Time = $\frac{\frac{Trial\ 1}{1000} + \frac{Trial\ 2}{1500} + \frac{Trial\ 3}{2000}}{3}$

Avg. Time for Original Manifold = 34.12 ms

Avg. Time for Modified Manifold = 28.39 ms

Chapter 6

Future Work

The features incorporated in to manifold so far are basic and provide limited functionality. But what makes the application exciting is that any developer can add as many new features as he desires and the type of these features can be left to his/her imagination, such is the design of manifold. A basic platform has been created that only has to be enhanced to make it a better application. This can be done by developing newer feature. While features are one part the other part would be to enhance the performance of the application in various ways. This section describes the scope of future work that can be done on manifold to enhance its features and performance.

There are certain features included in manifold which do not serve the purpose they were intended to. Below are described some of these features and how they can be incorporated. Newer features that can be incorporated in to manifold to provide the user with multitude of options have also been described. Several of these ideas can be imagined to be similar in functionality to those in popular applications such as Microsoft PowerPoint, Microsoft Word and Microsoft Paint.

6.1 Text Box

The *textbox* is a tool that can be used by a user to create text on the workspace. The current implementation of manifold on displays this option but does not implement

it. In the current implementation a user can draw a textbox across the workspace but can not write anything with in it. This is because the events generated by the keyboard do not have a *listener* with in the application. Future scope is to implement an `ActionListener` that listens to the users events from the keyboard and display them with in the *textbox* of the workspace. One can draw parallel ideas from the implementation of a similar *textbox* in Microsoft PowerPoint where the user can draw a *textbox* with in the workspace (in this case slides) and display text with in it. Added features to this would be the ability to stretch or compress the textbox while preserving the text with in the textbox as in MS PowerPoint. Additional features to the textbox would be to provide the user a set of property editors with which the user can modify its properties. Like, the user can assign color to the text, vary font, vary the fill and border color of the textbox.

6.2 Linker

The Linker is another dummy feature with in the current implementation of manifold. The linker is a line that has connection points at the ends of the line and stays connected to the glyph that you attach it to. Current implementation only displays this provision but does not implement it. However, ground work has been laid to facilitate this feature. `manifold.impl2D.tools.linker` is responsible for implementing this feature. While the class does not describe the process of linking two glyphs it does contain methods that listen to mouse actions and can recognize the node id of the glyph currently under the mouse cursor. This greatly reduces the work required to implement the linker feature.

Future scope of work which is required is to incorporate features that will allow the mouse to automatically grab the closest handle (connection point) of the first glyph to the mouse cursor when a manipulation cycle has begun. And similarly be able to grab the closest handle (connection point) to the mouse cursor on the second glyph in order to complete the manipulation cycle. This can be realized by highlighting the handles closest to the mouse cursor when the cursor is on the glyph and as it is moved over them. These highlighted handles indicate where the linker can be attached to. When glyphs joined by linkers are rearranged, the linkers remain attached to and move with the shapes. If either ends of a connector is moved, that end detaches from the glyph, and it can then be attach to another connection site on the same glyph or attached to another glyph. After the linker attaches to a handle, the linker should stay connected to the shapes no matter how the glyph is moved. An additional feature would be tor provide three types of linkers: straight, elbow (angled), and curved.

6.3 Manipulating Multiple Glyphs

The current implementation of manifold allows multiple selections of glyphs but does not allow the user to change their properties as a group. For example, one can draw a selection box around multiple glyphs but will not be able to drag this selection box around the work space. Only one glyph can be acted up on at a time. The option making this possible would be a useful feature.

6.4 New Property Editors

The list of new properties editors that can be incorporated can be enormous and best be left to the imagination of the developer. However, one feature of interest that we felt is important is to provide the user an option to move the glyph either to the background or foreground in a group of glyphs.

Another added feature would be to provide the user an option to Preview his actions before they are performed on the glyph. This would mean that the `manipulator` sends events to the domain *while* the object is being manipulated. Most editors allow "preview," through animation, and perform actions on the domain model only at the end of manipulation. This would be a cool feature.

6.5 Workspace Background Color

Most of the features covered so far are glyph specific. A new feature to change the "look" of the workspace will be a nice feature. A provision to fill the background with a fill color or placing a picture/clip art will provide the user with greater flexibility and options.

6.6 Thread Issues

If you anticipate working only with small domain models (with correspondingly small number of glyphs) and you are thread-thrifty, you may decide not to run `Display` in a separate thread. Rather, `Display` is asked to perform redraws within the current thread. How to redesign `Display` to be able to make this choice (threaded vs. non-

threaded) at runtime? Also, depending on the particular GUI toolkit used for the Manifold implementation, the toolkit may have mechanism for the frame rate control. This should be possible to exploit from `Display`.

Conclusion

Many of the user interface ideas that you may have seen articulated elsewhere, will be found implemented in Manifold. The Manifold framework presented here provides a domain-independent implementation of a presentation module. It translates the user's pointing gestures into action frames that are delivered to the underlying application domain. The conversational metaphor is exploited throughout the framework.

My work rid the application of the multitude of issues that inhibited it from functioning the way it was designed to. The changes to the application following the addition of the property viewer resulted in a fully functional application with working *line color editor* and *line width editor*.

I incorporated newer property editors to the property viewer panel that provides users enhanced options to change the properties of a selected glyph. These new properties included a *fill color editor* that fills the interior of a glyph with a user specified color and a *stroke editor* that changes the stroke of a selected glyph.

The `EventFrame` is responsible for translating the user's actions to a form that can be understood by the underlying domain model. I re-engineer the design of the `EventFrame` by changing its slot container to a `String` from a `Hashtable`. The changes to the `EventFrame` and the subsequent changes to the code to incorporate the new `EventFrame` resulted in enhanced performance of the application.

In conclusion, Manifold is currently completely error free and includes several newer property editors. The changes to the `EventFrame` enhanced the performance of frame transfer rate from a `manipulator` to the `Controller`.

References

1. Wikipedia: Graphical User Interface, Online at: http://en.wikipedia.org/wiki/Graphical_user_interface.

2. I. Marsic, *Manifold User Interface Framework,* Rutgers University, NJ, 2005.

3. I. Marsic*, An architecture for heterogeneous groupware application,* Proceedings of the 23rd IEEE/ACM International Conference on Software Engineering (ICSE 2001)*,* Toronto, Canada, pp. 475-484, May, 2001.

4. F.Flippo, A.Krebs and I.Marsic, *A framework for rapid development of multimodal interfaces, P*roceedings of the 5th International Conference on Multimodal Interfaces (ICMI 2003), Vancouver, B.C., Canada, pp. 109-116, November 2003.

5. B.A. Mayers, A Brief History of Human-Computer Interaction Technology, ACM interactions, 1998.

6. Wikipedia: Glyph, Online at: http://en.wikipedia.org/wiki/Glyph.

7. G. Krasner and S. Pope, *A cookbook for using the model-view-controller user interface paradigm in smalltake-8,* Journal of Object-Orienter Programming*,* vol. 1, no. 3, pp. 26-49, 1988.

8. E. Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison Wesley Longman, Inc, Reading, MA, 1995.

9. C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development,* 3rd edition, Prentice Hall PTR, Upper Saddle River, NJ, 1995.

10. *JavaBeans Component Architecture Webpage,* Online at: http://java.sun.com/products/javabeans/.

11. Sun Microsystems: Java 2D Tutorial, Online at: http://java.sun.com/docs/books/tutorial/2d/index.html.

12. J.M., Vlissides, UniDraw: A framework for building domain-specific graphical editors. [ed.] T. Lewis. *Object-Oriented Application Frameworks.* Greenwich, CT : Manning Publications, Co., 10, pp. 239-290, 1995.

13. J.M. Vlissides and M.A. Linton. Unidraw: A framwork for building domain-specific graphical editors. *ACM Transactions on Information Systems.* Vol. 8, 3, pp. 237-268, July 1990.

14. S.H. Tang and M.A. Linton, *Blending structured graphics and layout,* 7th ACM Annual Symposium on User Interface Software and Technology (UIST), Marina Del Ray, CA . pp. 167-173, November 1994.

15. S. Churchill, *Structured graphics in Fresco,* C++ Report, pp. 61-68. 3, March/April 1995.

16. Sun Microsystems: Class JComponent, Online at: http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JComponent.html.

17. Sun Microsystems: Class JPanel, Online at: http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JPanel.html.

18. Sun Microsystems: Class JLabel, Online at: http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JLabel.html.

19. Sun Microsystems: Interface Iterator, Online at: http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterator.html.

20. Sun Microsystems: Class NullPointerException, Online at: http://java.sun.com/j2se/1.4.2/docs/api/java/lang/NullPointerException.html.

21. Sun Microsystems: How To Write an Action Listener, Online at: http://java.sun.com/docs/books/tutorial/uiswing/events/actionlistener.html.

22. Sun Microsystems: Interface ActionListener, Online at: http://java.sun.com/j2se/1.4.2/docs/api/java/awt/event/ActionListener.html.

23. Sun Microsystems: How To Use Panels, Online at: http://java.sun.com/docs/books/tutorial/uiswing/components/panel.html.

24. Sun Microsystems: Class GridLayout, Online at: http://java.sun.com/j2se/1.4.2/docs/api/java/awt/GridLayout.html.

25. Sun Microsystems: Javax.Swing package, Online at: http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/package-summary.html.

26. Sun Microsystems: Class Color, Online at: http://java.sun.com/j2se/1.5.0/docs/api/java/awt/Color.html.

27. Sun Microsystems: Class JColorChooser. *Sun Java*, Online at: http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JColorChooser.html.

28. Sun Microsystems: How to Use Color Chooser, Online at: http://java.sun.com/docs/books/tutorial/uiswing/components/colorchooser.html.

29. Sun Microsystems: Class JButton, Online at: http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JButton.html.

30. Sun Microsystems: How to use Buttons, Checkboxes and Radio Buttons, Online at: http://java.sun.com/docs/books/tutorial/uiswing/components/button.html.

31. Sun Microsystems: Interface Shape, Online at: http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Shape.html.

32. Sun Microsystems: Class BasicStroke, Online at: http://java.sun.com/j2se/1.4.2/docs/api/java/awt/BasicStroke.html.

33. Sun Microsystems: Class JTextField, Online at: http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JTextField.html.

34. Sun Microsystems: Class JComboBox, Online at: http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JComboBox.html.

35. M.L. Minsky, *A Framework for Representing Knowledge,* MIT-AI Laboratory Memo, MIT, Cambridge, MA, pp 306, June 1974.

36. P.H. Winston, *Artificial Intelligence,* 3rd Edition, Addison-Wesley Publishing Company, Inc., Reading, MA, 1992.

37. Harold, Elliotte Rusty. *Processing XML with Java(tm): a guide to SAX, DOM, JDOM, JAXP, and TrAX, ISBN:* 0201771861, Addison-Wesley Publication, Inc, 2002.

38. Sun Microsystems: Class Hashtable, Online at: http://java.sun.com/j2se/1.4.2/docs/api/java/util/Hashtable.html.

39. Sun Microsystems: Class String, Online at: http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html.

40. Sun Microsystems: Class Object, Online at: http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html.