

©2010

Harsh Yadav

ALL RIGHTS RESERVED

MANIFOLD: A MULTIMODAL GENERIC USER INTERFACE

by

HARSH YADAV

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 and Computer Engineering

written under the direction of

Dr. Ivan Marsic

and approved by

New Brunswick, New Jersey

October, 2010

ABSTRACT OF THE THESIS

Manifold: A Multimodal Generic User Interface

By HARSH YADAV

Thesis Director:

Dr. Ivan Marsic

A User Interface (UI) is generally tightly coupled to the business logic of the application, and a great deal of modification is required to model it to separate the application logic. Manifold is an answer to such an issue. It offers a generic user interface framework where different application designers could develop separate application logic using the same Manifold front-end. Manifold employs the Model-View-Controller (MVC) design pattern to achieve its generic nature by providing separation of concerns between the UI and the application domain. The event frame concept incorporated on Manifold allows translation of user events to operations on application data via a central Controller object. The Controller object connects the Manifold front-end to the back-end application domain. This approach highly expedites the development of new features on the user interface, which remains decoupled from the rest of the application. It also allows building different applications on different servers and visualizing them on the Manifold UI.

This thesis focuses on the enhancement of the Manifold framework both from the UI- and application-development perspective. New features based on industry standards are incorporated and the shortcomings present in the previous versions are eliminated to enhance the user-experience with Manifold. The flexibility of the Manifold framework allows development of applications with separate application logic and *attaches* them directly with the Manifold UI.

The thesis developed applications and algorithms to visualize Manifold objects as fields in a relational database. The relational database approach allows saving the UI objects into a hierarchical data structure, rather than saving them as such on the file system. This provides added benefits of persistent information about objects, allowing data-analysis to be performed on them at various steps. This approach allowed extending the use of Manifold to areas such as text-, pattern- and speech recognition. Text recognition allows user to generate graphical objects (“glyphs”) on Manifold by typing in plain text data. Pattern recognition allows providing users with visual feedback if a collection of graphics in Manifold could be recognized as a particular pattern class. Speech Recognition allows the user to issue commands to the Manifold UI to perform specific tasks, which otherwise require manual manipulation.

The newly developed features are analyzed for their structural and code complexity with direct repercussions to their re-use in future designs and performance optimization obtained by employing the best software engineering practices. The new features are evaluated with respect to their performance in the Manifold framework.

Acknowledgement and/or Dedication

I am particularly grateful to my supervisor, Professor Ivan Marsic who despite of his busy schedules always offered prompt, wise and constructive feedback and displayed high professionalism and graciousness to remain dedicated to the research. Professor Marsic's plethora of knowledge, insight and untiring work ethic had always motivated me and will continue to be a source of inspiration to me.

A special thanks to my brother, Tarun Yadav, for his unfailing support and constructive criticism for both in my personal and professional front.

The long, hard process of completing a thesis would have been completely impossible without the support of many friends and colleagues at Rutgers.

This thesis would not have been possible without the excellent architecture and calm environment at the Centre for Advanced Information Processing (CAIP).

Table of Contents

Abstract	ii
Acknowledgements	iv
List of Tables	xi
List of Illustrations	xii
CHAPTER 1.....	1
INTRODUCTION	
.....	1
1.1 MANIFOLD CORE FRAMEWORK.....	3
1.1.1 <i>Model</i>	5
1.1.2 <i>Controller</i>	6
1.1.3 <i>View</i>	6
1.2 MANIFOLD USER INTERFACE.....	7
1.3 BRIEF INTRODUCTION TO MY WORK	8
1.3.1 <i>Developing Backend Applications</i>	9
1.3.2 <i>New Features and Enhancements</i>	9
1.3.3 <i>Multimodal Interaction Techniques</i>	10
1.3.4 <i>Glyphs and Tools</i>	12
1.3.5 <i>Menu Bar</i>	12
1.3.6 <i>Property Editors</i>	13
1.4 THESIS ORGANIZATION.....	13
CHAPTER 2.....	15
MANIFOLD CORE FRAMEWORK	
.....	15

2.1	INTRODUCTION	15
2.2	MODEL	16
2.2.1	<i>Manifold Glyph Design</i>	18
2.3	VIEW	19
2.4	CONTROLLER	20
2.4.1	<i>Manipulator</i>	21
2.4.2	<i>Controller Communication</i>	22
CHAPTER 3.....		24
DEVELOPING BACKEND APPLICATIONS		
.....		24
3.1	INTRODUCTION	24
3.2	COMMUNICATION PROTOCOLS	25
3.2.1	<i>Information Exchange: Formats</i>	26
3.2.2	<i>Information Exchange: Medium</i>	28
3.3	EXAMPLE APPLICATION: PATTERN RECOGNITION	29
3.3.1	<i>Introduction: Pattern Recognition</i>	30
3.3.2	<i>Design: Pattern Recognition</i>	32
3.3.3	<i>Data Flow in Pattern Recognition System</i>	34
3.3.3.1	<i>Unknown Object for Pattern Recognition System</i>	35
3.3.3.2	<i>Feature Analysis: Parameter Extraction</i>	38
3.3.3.3	<i>Feature Analysis: Feature Extraction</i>	39
3.3.3.4	<i>Pattern Classification</i>	41
3.3.3.5	<i>Rendering on Manifold</i>	44
CHAPTER 4.....		47

ENHANCEMENTS AND MULTIMODAL INTERACTION TECHNIQUES

.....	47
4.1 INTRODUCTION	47
4.2 NEW FEATURES AND ENHANCEMENTS	48
4.2.1 <i>Editing Multiple Glyphs</i>	48
4.2.2 <i>Poly Glyph</i>	50
4.2.3 <i>Glyph Selection</i>	51
4.2.4 <i>Manifold Configuration</i>	52
4.2.5 <i>Correcting Bounding Shapes</i>	54
4.3 MULTIMODAL INTERACTION TECHNIQUES.....	54
4.3.1 <i>Text Recognition</i>	55
4.3.1.1 <i>Design: Text Recognition</i>	56
4.3.1.2 <i>Implementation: Text Recognition</i>	58
4.3.1.3 <i>Rendering Text Recognition Output on Manifold</i>	63
4.3.2 <i>Speech Recognition</i>	63
4.3.2.1 <i>Speech Recognition Application in Manifold</i>	64
CHAPTER 5.....	66

GLYPHS AND TOOLS

.....	66
5.1 INTRODUCTION: TOOLS, MANIPULATORS AND CONTROLLER	66
5.2 GLYPHS AND VIEWERS	68
5.3 PREVIOUSLY NON-FUNCTIONAL GLYPHS AND TOOLS	70
5.3.1 <i>Text Glyph</i>	70
5.3.2 <i>Zoomer</i>	73
5.4 NEW GLYPHS AND TOOLS	76

5.4.1	<i>Image Glyph</i>	76
5.4.2	<i>Grouper</i>	78
5.4.2.1	<i>Design: Grouper Tool</i>	80
5.4.2.2	<i>Interpreting Grouped Glyphs</i>	84
5.4.3	<i>Un-Grouper</i>	85
5.4.4	<i>Pinner</i>	86
CHAPTER 6		88
MENU BAR		
.....		88
6.1	INTRODUCTION	89
6.2	DESIGN: MENU BAR	90
6.3	FILE MENU	93
6.3.1	<i>New Workspace Menu Item</i>	93
6.3.2	<i>Open Document Menu Item</i>	94
6.3.3	<i>Save Selection(s) Menu Item</i>	96
6.3.4	<i>Save Document Menu Item</i>	102
6.4	EDIT MENU	103
6.4.1	<i>Select All Menu Item</i>	104
6.4.2	<i>Select None Menu Item</i>	105
6.5	VIEW MENU	105
6.5.1	<i>Full Screen Menu Item</i>	105
6.5.2	<i>Minimize Menu Item</i>	106
6.5.3	<i>Map Viewer Menu Item</i>	106
6.5.3.1	<i>Design: Map Viewer</i>	109
6.6	INSERT MENU	113

6.6.1	<i>Geometric Figure Menu</i>	114
6.6.1.1	<i>Rectangle Menu Item</i>	114
6.6.1.2	<i>Ellipse Menu Item</i>	116
6.6.1.3	<i>Line Menu Item</i>	116
6.6.2	<i>Image Menu Item</i>	116
6.6.3	<i>Custom Glyph Menu Item</i>	117
6.6.4	<i>Smart Art Menu Item</i>	121
CHAPTER 7.....		123
PROPERTY EDITORS		
.....		123
7.1	INTRODUCTION	123
7.2	DESIGN: PROPERTY EDITOR	126
7.3	NEW PROPERTY EDITORS	127
7.3.1	<i>Text Editor</i>	127
7.3.2	<i>Font Editor</i>	129
7.3.3	<i>Image Editor</i>	133
CHAPTER 8.....		136
COMPLEXITY AND PERFORMANCE		
.....		136
8.1	DESIGN COMPLEXITY	136
8.1.1	<i>Structural Analysis</i>	137
8.1.2	<i>Excessive Structural Complexity</i>	141
8.1.3	<i>Code Analysis</i>	144
8.2	PERFORMANCE	154

8.2.1	<i>Application Loading Time</i>	154
8.2.2	<i>Performance: Glyph Saving</i>	157
8.2.3	<i>Performance: Saved Glyph Retrieval</i>	158
8.2.4	<i>Performance: Grouper Tool</i>	158
8.2.5	<i>Performance: Custom Glyph Insertion</i>	160
CHAPTER 9.....		161
DISCUSSION AND FUTURE WORK		
.....		161
9.1	FUTURE WORK.....	164
9.1.1	<i>New Workspace</i>	164
9.1.2	<i>Keyboard Listeners</i>	165
9.1.3	<i>New Features</i>	165
9.1.3.1	<i>Undo/Redo Features</i>	165
9.1.3.2	<i>New Glyph Types</i>	167
REFERENCES		
.....		169

Lists of tables

Table 1: Manifold classes in the Glyph Inheritance hierarchy	17
Table 2: SQL stored procedures for the process of pattern recognition	44
Table 3: SQL stored procedures for the process of text recognition	62
Table 4: Manifold Tools and their Functionality	68
Table 5: Manifold Glyphs and their Representation on the Viewer	69
Table 6: Changes made in Manifold classes to support Grouper tool.	85
Table 7: New Manifold classes for implementing a functional Menu Bar.....	91
Table 8: Manifold classes used to layout the Map Viewer	110
Table 9: Current Manifold Structure Statistics Summary.....	139
Table 10: Top 5 Manifold classes and packages in terms of LOC	146
Table 11: Code Complexity Metrics.....	146
Table 12: High Code Complexity methods in package manifold.....	149
Table 13: High Code Complexity methods in package manifold.swing	150
Table 14: High Code Complexity methods in package manifold.swing.editors	150
Table 15: High Code Complexity methods in package manifold.impl2D.....	151
Table 16: High Code Complexity methods in package manifold.impl2D.glyphs.....	152
Table 17: High Code Complexity methods in package manifold.impl2D.tools.....	152
Table 18: Comparison of current vs. previous version of Manifold.....	155
Table 19: Method Call Hierarchy for Save Glyph Menu Item	157
Table 20: Method Call Hierarchy for Insert Menu Item.....	158
Table 21: Method Call Hierarchy for Grouper Tool.....	159
Table 22: Method Call Hierarchy for Custom Glyph Insertion.....	160

List of illustrations

Figure 1: Manifold User Interface Abstraction and the MVC design pattern ([4])	4
Figure 2 Manifold User Interface	7
Figure 3: Manifold (high-level) architecture	16
Figure 4: Composite Design Pattern	17
Figure 5: Manifold UI Interaction with other applications. Notice same protocols are used for interacting with multimodal interaction techniques as for other backend applications	26
Figure 6: A conventional pattern recognition system	31
Figure 7: Database diagram for Manifold Pattern Recognition.....	32
Figure 8: Data flow in pattern recognition system.....	34
Figure 9: Higher Level data flow between Manifold and Pattern recognition system	35
Figure 10: User trying to visualize a pattern via a number of glyphs (unknown object for Pattern Recognition system) on Manifold.	37
Figure 11: Parameter $x(1)$	38
Figure 12: Parameter $x(2)$	38
Figure 13: Parameter $x(3)$	39
Figure 14: Parameter $x(4)$	39
Figure 15: (a) Glyph's prototype as represented in its local coordinate system. (b) Glyph transformed in the global coordinate system: positioned at (3, 5), width scaled to 6 and height to 4, and rotated by $\theta = 30^\circ = \pi/6$	40
Figure 16: Feature vector y (with dimensionality $m \leq p$)	41
Figure 17: JSON object send to Manifold over the communication channel	43

Figure 18: Pattern simple switch is classified, but not matched (red color of ellipse)	46
Figure 19: Circuit is complete (line connects two rectangles), and the pattern is matched (yellow color of ellipse)	46
Figure 20: Visualization of a battery-bulb pattern on Manifold	46
Figure 21: A battery bulb pattern classified and matched successfully (as all circuit elements are well connected)	46
Figure 22: Empty Property Viewer on selection of multiple glyphs in previous versions of Manifold	49
Figure 23: Editing properties of multiple glyphs in current Manifold version (Fill Color property being edited here)	49
Figure 24: The Manifold Configuration Wizard, welcome text	53
Figure 25: The Manifold Configuration Wizard, selecting an XML file	53
Figure 26: The Manifold Configuration Wizard, configuring application with selected XML files	54
Figure 27: Database schema for Text Recognition	57
Figure 28: Flow chart depicting the entire process of Text recognition	59
Figure 29: Word tokens generated as a result of preprocessing. Notice it doesn't contain any noise words	62
Figure 30: Bigrams from the preprocessed output	62
Figure 31: Final Glyph Profile that can be passed on to Manifold	62
Figure 32: The main input and outputs of a Tool/Manipulator component	66
Figure 33: Original Glyph without applying the Zoomer Tool	75

Figure 34: Zoomed In Glyph as a result of applying the Zoomer Tool to glyph in Figure 33.....	75
Figure 35: Use Case Diagram for Grouper	80
Figure 36: UML Sequence Diagram showing the grasp, manipulate, effect cycle of Grouper Tool.....	83
Figure 37: Five glyphs drawn independently on the Manifold Viewer.	84
Figure 38: The glyphs from Figure 37 after grouping.	84
Figure 39: The independent glyphs can now be treated as a single smart art (a smiley here). Notice the connectors become visible when you scroll over the object.	84
Figure 40: A pinned glyph of type rectangle	87
Figure 41: Glyph's translation being disabled at the pinned connector.....	87
Figure 42: The pinned glyph could still be rotated at the pinned connector.....	87
Figure 43: XML definition of Manifold Menu Bar	90
Figure 44: Composition of Menu Bar: (a) The screen rendering; (b) The UML class diagram	92
Figure 45: Output when the user clicks on the Open Document Menu item.....	95
Figure 46: Database diagram showing the table relationship to save glyphs (leaf- or poly glyphs) or document.	100
Figure 47: A JOptionPane [18] appears when the user presses the Save Selection(s) menu item.	101
Figure 48: New entry created in table Glyph. Notice the parentGlyphId is NULL as this is not a poly-glyph.	101

Figure 49: New entry created in table GlyphProfile. Notice the glyph properties and transform (tx, ty, theta, xs, ys) stored as text attributes in the table.	101
Figure 50: The Manifold Map Viewer displaying a graph connecting various locations on the map using Manifold glyphs.....	109
Figure 51: Composition of Map Viewer: (a) The screen rendering; (b) The UML class diagram	111
Figure 52: Sequence Diagram showing ViewerMapImpl<init> and ViewerMapImpl.actionPerfrommed cycles	112
Figure 53: The Manifold Insert Menu	113
Figure 54: List of saved glyphs of type “rectangle” retrieved from the database, along with its profile (properties and transform).	115
Figure 55: Manifold Custom Glyph. The user enters input in a JOptionPane.....	118
Figure 56: A new glyph gets created and drawn on the Manifold viewer as a result of Text Recognition.....	119
Figure 57: Manifold interaction with the remote server through SOAP web service	120
Figure 58: Example of a property editing dialog box. Property editors allow editing the property values. [4]	124
Figure 59: Composition of a property editor dialog box: (a) The screen rendering; (b) The UML class diagram [4]	125
Figure 60: The Text Editor. When the user presses the return (enter) key, the new text is rendered on the Text glyph.	129
Figure 61: The Font Editor. Notice the new font size and style that gets rendered on the Text glyph.	133

Figure 62: Image Editor: When the user clicks on the button, a file browser appears where the user can select the new Image.....	135
Figure 63: New image being rendered as a result of user's new selection from the file browser.....	135
Figure 64: Manifold Core Structure.....	137
Figure 65: The “onion” structure of Manifold packages	138
Figure 66: Package level Architecture of Manifold (The arrows specifies use, for e.g. manifold uses util).....	138
Figure 67: Statistics of the connectivity of all the classes in the current Manifold implementation.	140
Figure 68: Pie Chart showing Tangles in Manifold design	141
Figure 69: Fat vs. non-Fat methods in Manifold	143
Figure 70: Pie Chart showing the contributors to the Structural Level Complexity of Manifold.....	144
Figure 71: Methods with different levels of complexity in package manifold.....	147
Figure 72: Methods with different levels of complexity in package manifold.swing	147
Figure 73: Methods with different levels of complexity in package manifold.swing.editors.....	147
Figure 74: Methods with different levels of complexity in package manifold.impl2D..	147
Figure 75: Methods with different levels of complexity in package manifold.impl2D.menuItems	148
Figure 76: Methods with different levels of complexity in package manifold.impl2D.glyphs	148

Figure 77: Methods with different levels of complexity in package manifold.impl2D.tools	148
Figure 78: Methods with different levels of complexity in package manifold.data	148
Figure 79: Comparison of Links vs. Fat method in Manifold. Notice that the core classes are not fat.	153
Figure 80: Comparison of loading times of two versions of Manifold.....	156

Chapter 1

Introduction

The study of the relationship between humans and computers has quickly become one of the most dynamic and significant fields of technical investigation. Interdisciplinary by definition, *Human-Computer Interaction* (HCI) impacts nearly every area of our lives. HCI arose as a field from intertwined roots in computer graphics, operating systems, human factors, ergonomics, industrial engineering, cognitive psychology, and the systems part of computer science [1]. Today, the sales of computers are more tied to the quality of the interfaces than in the past.

With the gradual evolution of standardized interface architecture from hardware support of mice to shared window systems to "application management layers", researchers and designers have begun to develop specification techniques for user interfaces and testing techniques for the practical production of interfaces. *Graphical User Interfaces* (GUI) [3] and HCI are largely affected by certain forces that also affect the nature of future computing. These forces include:

- Decreasing hardware costs leading to larger memories and faster systems.
- Miniaturization of hardware leading to portability.
- Reduction in power requirements leading to portability.
- New display technologies leading to the packaging of computational devices in new forms.

- Assimilation of computation into the environment (e.g., VCRs, microwave ovens, televisions).
- Specialized hardware leading to new functions (e.g., rapid text search).
- Increased development of network communication and distributed computing.
- Increasingly widespread use of computers, especially by people who are outside of the computing profession.
- Increasing innovation in input techniques (e.g., voice, gesture, text, pen), combined with lowering cost, leading to rapid computerization by people previously left out of the "computer revolution".
- Wider social concerns leading to improved access to computers by currently disadvantaged groups (e.g., young children, the physically/visually disabled, etc.).

This thesis mainly focuses on GUI's, and increasing innovation in input techniques (voice, text) on the *Manifold Framework* [4]. Manifold's focus is on conversational human-computer interaction, which may include, but is not limited to, manual gestures, spoken language, etc. A *User Interface* (UI) [2] acts as a liaison between the high-level and low-level language formats, as understood by human beings and computers respectively. However, a UI is generally highly coupled to its particular application domain and a great deal of work is required to remold it to a different application [4]. Manifold is an answer to such a problem to develop an "application-independent generic user interface", such that the same UI could be easily "detached" and "attached" to different applications, running parallel with respect to each other. The first version of Manifold appeared in [5] and was also based on the work [6, 76].

There has been certain amount of work been done in the area in developing similar applications. Unidraw [58] provided a C++ framework for custom graphical editing applications. Girders [88] utilizes MVC functionality for a multi-tiered software application communicating through SOAP web services. Strandz [89] provides middle layer that connects a Java Swing UI with Data Objects (MVC). It provides application development that is easy to modify and resilient to changes to the user interface (UI) toolkit or database access software they use.

The idea behind Manifold's development is to provide different developers to work on different modules independently:

- *Interaction Developer* could define new physical manipulations.
- *Visualization Developer* could define new UI elements (e.g. glyphs).
- *Multi-modality Developer* could introduce use of exotic input devices with existing manipulations.
- *Application Developer* could specify business logic independent of UI.

In order to understand the current work, it is critical to understand the architecture of Manifold. This introduction provides an overview of the Manifold framework. The chapters that follow will provide a serious, detailed explanation of these problems and the solutions employed.

1.1 Manifold Core Framework

The design of Manifold has been done keeping in mind some of the best software engineering practices. Some of these include design patterns; reduced code dependency,

separation of application and domain logic, etc. The reader is advised to read the Manifold text [4] for a better understanding of the framework. Here I would be covering a thorough, but brief overview of the core Manifold framework.

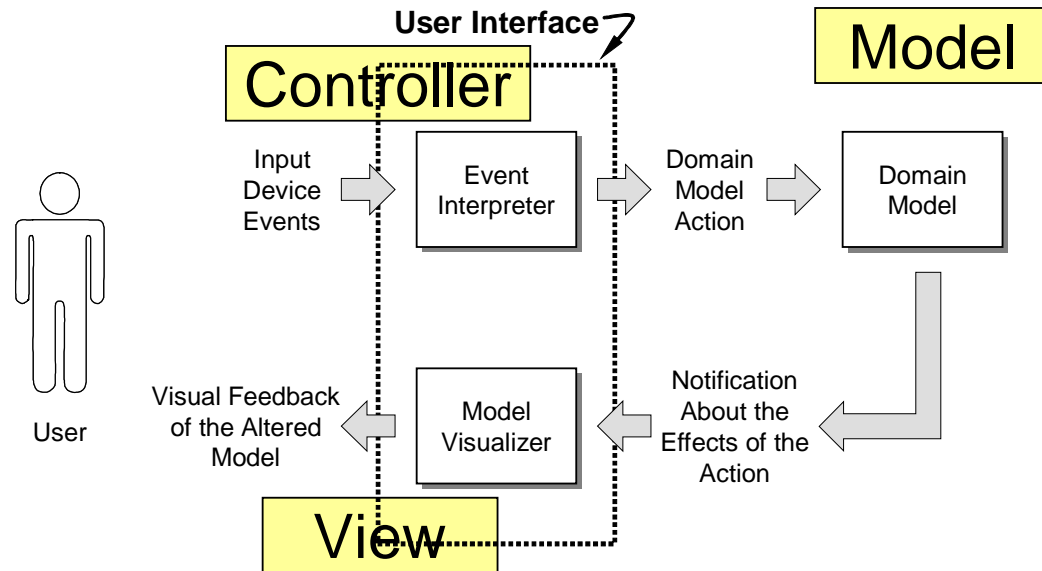


Figure 1: Manifold User Interface Abstraction and the MVC design pattern ([4])

The core functionality of Manifold is defined by the *Model View Controller* (MVC) design pattern [8, 57]. MVC provides Manifold the capability to be developed in modules with minimum coupling between them. It allows the separation of the application and the domain logic, as highlighted in Figure 1.

The user actions are interpreted by the *Controller* and are passed on to the *Model* for manipulation, which responds back by providing visual notifications to the user through the *Viewer*. The *Viewer* implements the *Observer* design pattern [8] by reading the subject state upon being notified about the state changes.

1.1.1 Model

The domain model reads the input device events to process these commands and perform the required action. These two steps are abstracted as the lower and upper arms in Figure 1, respectively.

The elements of the domain model are visualized through a *Glyph* which is a visual representation corresponding to a model data element in a domain model. The name "glyph" is borrowed from typography to connote simple, lightweight objects with an instance-specific appearance [56]. The key purpose of Glyph is to implement the *Composite* design pattern [8], so to be able to hierarchically compose Glyphs into more complex figures.

The reader should keep in mind that glyphs represent merely a visual appearance of the underlying elements of the domain model. The actual implementation of Manifold may be built using a graphics toolkit that already has an equivalent of Glyph, in which case it does not need to use this class (due to these reasons Manifold classes are not dependent on this class). The container of glyphs uses the tree data structure through which it becomes easier to manipulate actions like *Add*, *Delete*, and *Modify*. For further understanding of the glyphs including the glyph design, state caching, shadow glyphs, dynamics and visualization, rendering and their implementation in Manifold, see Chapter 2, section 2.2.1.

1.1.2 Controller

The *Controller* is responsible for encapsulating the semantics of user interaction with the application. The user handling of input device(s) generates interaction events, which need to be translated to the domain model. These input events may come from focusable devices such as keyboard or voice, and positional devices, such as the mouse or pen. For example, the user's activity of depressing 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. The selected tool "knows" which one of these is currently in effect. The design espoused here is inspired by *Unidraw* [58] and *Fresco* [59]. To further understand the working of Controller in Manifold, the reader must refer to Chapter 2, section 2.4.

The earlier versions of Manifold considered events from only positional devices. A major contribution of my work includes manipulation of events for text and voice, which is discussed in Chapter 4.

1.1.3 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. It also redraws the affected part as soon as a change in the model occurs. 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. An application of implementing these multiple

viewports is discussed in Chapter 6, section 6.5.3, where two different viewers, a map and a glyph, interchangeably support different events to fulfill the domain specific requirements.

1.2 Manifold User Interface

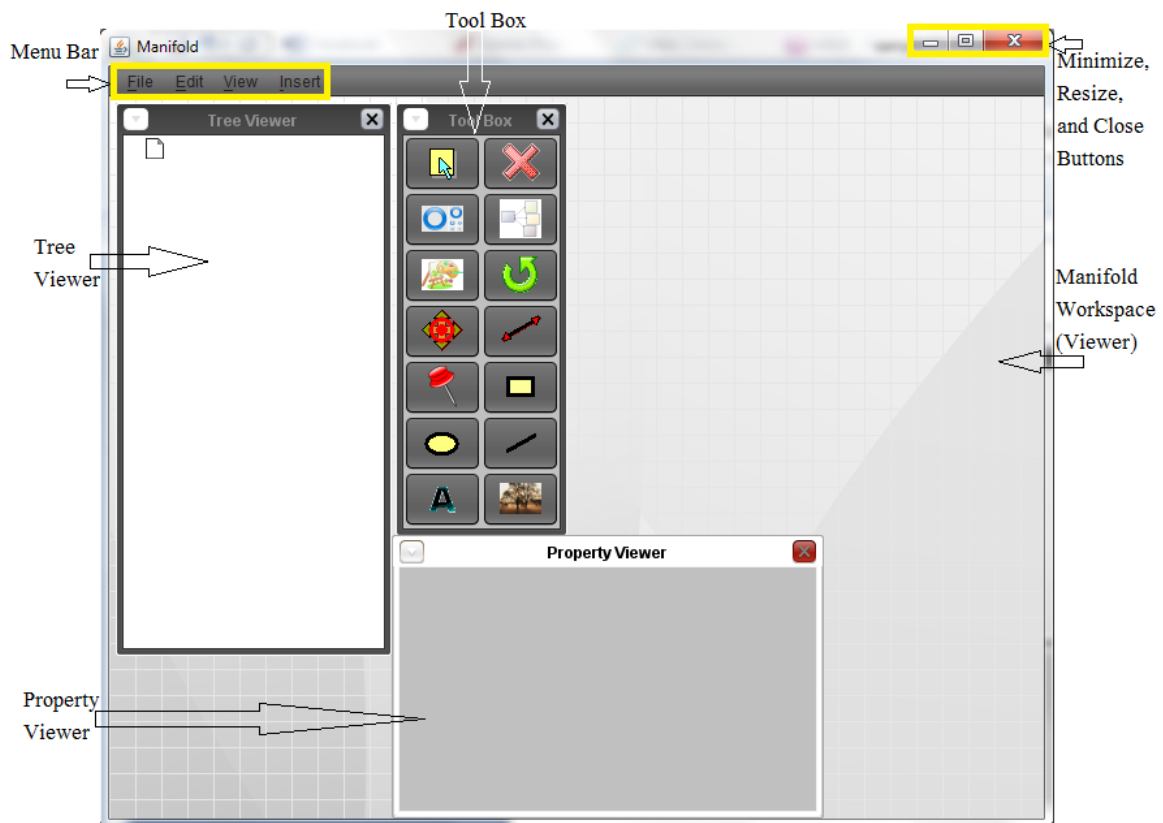


Figure 2 Manifold User Interface

Figure 2 shows the Manifold User Interface in its current implementation. It has the following main components that will be discussed in detail throughout the text:

- *Workspace* defines the area where the user can create and modify the properties of the glyphs such as translation, rotation, and appearance.

- *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 node 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 from the *Tree Viewer*.
- *Tool Box* provides the user a set of tools to perform manipulations on the glyph objects including their creation, deletion, rotation, zooming, grouping, un-grouping, pinning, linking, and selecting. Chapter 5 elaborates various tools and their functionalities.
- *Menu Bar* provides *File*, *Edit*, *View* and *Insert* menus to perform actions with respect to glyphs (saving, inserting, etc.) and the workspace (open new, change the viewer). Chapter 6 discusses the implementation of these menus in detail.
- *Property Viewer* allows the user to edit certain attributes of 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. Chapter 7 discusses the Property Viewer in detail.

1.3 Brief Introduction to my work

My work on Manifold can be categorically identified into the following specific areas:

1.3.1 Developing Backend Applications

To utilize the generic nature of Manifold, it was necessary to lay down the grounds for developing example backend applications that application programmers could develop and visualize them on Manifold. Manifold simply provides a visual feedback to the user by initiating a separate backend application to perform semantic processing of data, where the semantics are application specific. This has been explained by developing a simple example pattern recognition application.

Pattern Recognition that includes identifying a particular pattern (like simple switch), and generate appropriate output on the Manifold framework. For instance, if a user draws two rectangles connected by a line and an ellipse alongside it, it would have no meaning to Manifold, but for the user it could mean a simple switch-bulb connection, and the bulb (ellipse) should lighten (become yellow) when the wire (the line) connects the two boxes (rectangles) (Chapter 3, section 3.3).

The highly de-coupled nature of these techniques with Manifold shows its generic nature. The corresponding details and how Manifold uses them without being coupled with them had been provided in Chapter 3. The reader is advised to go through the same to appreciate the generic nature of Manifold.

1.3.2 New Features and Enhancements

With the availability of a large number of user-interface tools, there was a need to enhance some of the features in Manifold, which were envisioned in the previous versions. These are listed as below:

- Allowing editing of multiple glyphs with a single mouse click, by grouping the property editors common to multiple selected glyphs (Chapter 4, section 4.2.1).
- Implementation of *Poly-Glyph* feature based on the Composite Design Pattern [8] in Manifold by altering the parent-child properties (implemented as a part of the *Grouper* tool) (Chapter 4, section 4.2.2).
- Allow selection of `Line2D` (line and linker) glyphs that are rendered differently as compared to the rectangular glyphs by drawing a selection box around them on the Manifold viewer (Chapter 4, section 4.2.3).
- Allowing configuration of Manifold before running it, by providing a *Wizard* [11] to select the necessary XML [12] files to configure Manifold as per user requirement which could enhance the speed by limiting the functionality as per the needs of different users (Chapter 4, section 4.2.4).
- Correcting *bounding shapes* for text and image glyph, as the *highlighter* (a shadow glyph) was drawn separately from the main glyph in the earlier versions (the bounding shape and highlighter were misplaced and it was necessary that both of them coincide, as highlighter allows manipulation of underlying glyph by shadowing it) (Chapter 4, section 4.2.5).

To have an in-depth analysis of these features and enhancements, and why it was necessary to implement them, the reader is advised to read Chapter 4.

1.3.3 Multimodal Interaction Techniques

The growth and enhancement of *Human Computer Interaction* (HCI), makes it pertinent to include new *interaction techniques* in any modern user-interface. Doing so

become even more important if you are developing a generic user interface, where different users have to work in different environments with different requirements.

Majority of my work had been on developing these techniques with respect to the Manifold framework. Some of the techniques that I had worked on in this respect are:

- *Text Recognition* that includes parsing a raw junk of text to provide a meaningful interpretation to Manifold, allowing it to take necessary action. For instance if you input a raw text that has certain accompanying action (For e.g. “Draw a rectangle of height 25 and width 50 with rotation of 45 degree”), it could be understood by Manifold (and the corresponding rectangle be drawn on Manifold viewer space) (Chapter 4, section 4.3.1).
- *Speech Recognition* that utilizes the speech recognition techniques to provide input to Manifold. For instance if a user speaks “Draw a rectangle of height 25 and width 50 with rotation of 45 degree”, the spoken input could be understood by Manifold (and as a result the rectangle is drawn on the viewer) (Chapter 4, section 4.3.2).

These applications do not form a part of Manifold, but are available as utility applications that could be *attached* with Manifold as per the choice of an application programmer. All these techniques had been implemented on a remote server with separate application logic, and had been used in Manifold by interacting through a set of communication protocols as discussed in Chapter 3, section 3.2.

1.3.4 Glyphs and Tools

One of my tasks was to implement some of the non-functional and anticipated glyphs and tools from the earlier versions of Manifold. These were essential in improving and enhancing the core functionality of the Manifold framework that could make it at par with the modern GUI tools available, adhering to its “generic nature”. To enumerate, these tools were *Zoomer* (allows zooming in and out of canvas and glyph; Chapter 5, section 5.3.2), *Grouper* (allows grouping a number of selected glyphs to a single smart-art, Chapter 6, section 6.6.4; Chapter 5, section 5.4.2), *Un-Grouper* (allows splitting the grouped smart-art into independent glyphs from which it was generated; Chapter 5, section 5.4.3), *Connector* viz. *Pin* and *Slot* (allows connecting two glyphs with different connection semantics). The previously anticipated glyphs were *Text* (allows writing text on the Manifold workspace; Chapter 5, section 5.3.1) and *Image* (allows rendering of images on the Manifold workspace; Chapter 5, section 5.4.1). These glyphs and tools along with their implementation are discussed in details in Chapter 5.

1.3.5 Menu Bar

A *menu bar* is a horizontal strip where a list of available computer *menus* is housed for a certain program. A menu provides a space-saving way to let the user choose one of several options.

Since Manifold is a GUI there was a need to incorporate a functional menu bar that could allow the users to “interact” with the application in a better and useful way. Similar to other GUI applications like *MS PowerPoint* [9] and *Adobe Photoshop* [10], the

most common menu items anticipated for Manifold were *File*, *Edit*, *View*, and *Insert*. The *File* menu allows the user to open/save a document, save selection(s). The *Edit* menu allows the user to pass actions like undo/redo, make or clear selections to the application domain. The *view* menu allows the user to change the layout of the workspace like making it full screen, minimizing, opening a map viewer. Finally, the *insert* menu allows the user to insert objects and images on the workspace. These menus were anticipated in the previous versions of Manifold, however were non-functional. To make the interaction with Manifold user friendly and easy, I worked on making some of these menu items functional. The Manifold menu bar is discussed in detail in Chapter 6.

1.3.6 Property Editors

The addition of new glyphs demanded new property editors pertaining to each glyph. For example, *Font* (face, style, size) and *Text* editors were required for the Text Glyph; *Image* editor was added to the Image Glyph. These property editors were incorporated within the Property Viewer of the Manifold. These editors and their implementation are discussed in details in Chapter 7.

1.4 Thesis Organization

This thesis is mainly concerned with enhancing the Manifold Framework. It is organized as follows:

Chapter 1: This chapter gives an overview of Manifold, its architecture and the high level overview of its components. It also covers in brief the overview of my work, and the improvements/features added to Manifold framework.

Chapter 2: This chapter gives a detailed architectural overview of the core Manifold framework and design.

Chapter 3: This chapter describes how the development of example backend applications could be carried out to visualize them on the Manifold UI (generic nature).

Chapter 4: This chapter discusses certain new features and enhancement to some previously implemented features in Manifold. It also discusses the development of multimodal interaction techniques viz. Text and Speech Recognition for using Manifold.

Chapter 5: This chapter gives a detailed explanation of my work to implement new tools and glyphs for the Manifold UI.

Chapter 6: This chapter gives a detailed explanation on the implementation of the Manifold menu bar and the corresponding menus and menu items.

Chapter 7: This chapter describes the addition of new property editors in Manifold UI.

Chapter 8: This chapter provides the complexity and performance analysis of Manifold's code and design.

Chapter 9: This chapter concludes the whole thesis and proposes future work.

Chapter 2

Manifold Core Framework

Chapter 1 discussed the basic overview of the Manifold architecture that uses the MVC design pattern at its core. Manifold employs some of the best software engineering practices in order to provide a sleek and intuitive user interface. The development of the Manifold framework has been iterative and incremental where each version provided newer features and improvements over the older. This had been possible due to a highly stable and concrete Manifold core. Thus, in order to better understand my work, it is important that one understands the Manifold core architecture. This chapter describes certain critical aspects of the architecture and provides a “somewhat” in depth analysis of the topics that would be needed in understanding my work. The reader is highly advised to first read the documentation provided by Dr. Ivan Marsic [4] and then refer to the current description while going through different parts of this text for increased clarity.

2.1 Introduction

Manifold employs Sun Microsystems Java (TM) Technology [63] as its primary coding language to design the interface and its underlying application. This makes the system platform independent and highly stable to work on. The core framework uses the MVC design pattern as discussed in Chapter 1. This can be visualized at a high level in Figure 3.

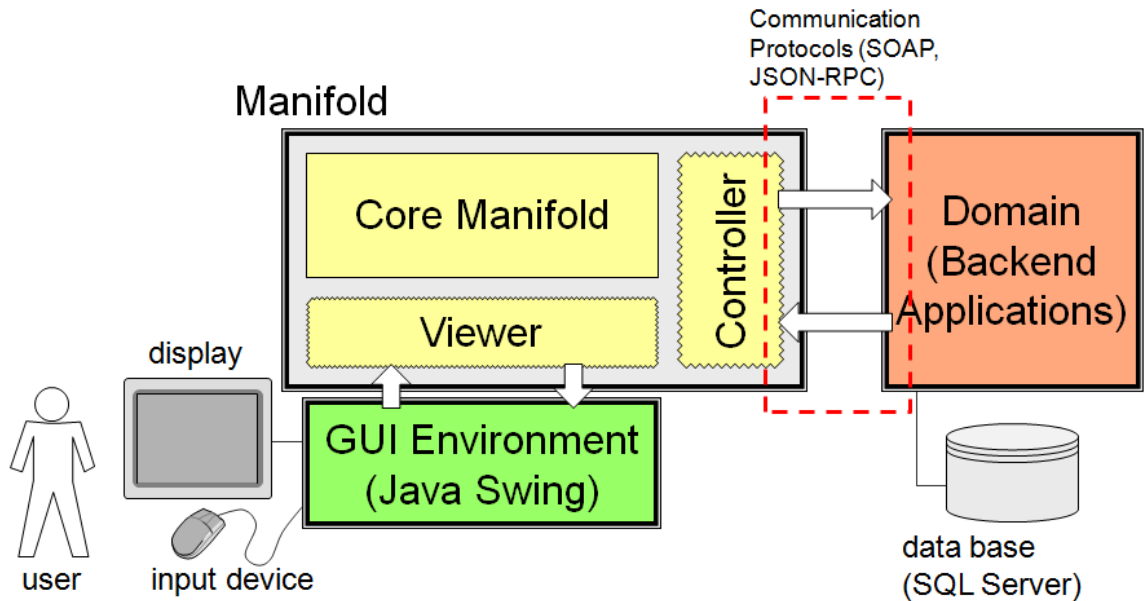


Figure 3: Manifold (high-level) architecture

The MVC design pattern provides a highly de-coupled sequence of flow by accepting the user events from an input device on the UI, interpreting these events using the Controller, and providing the visual feedback by determining how the content would be altered as a result of the interpreted events in the Model. The benefits of the *Model-View-Controller* (MVC) design pattern were first discussed in [61] and the reader should also check [62].

2.2 Model

As discussed in Chapter 1, section 1.1, Manifold uses glyphs to visualize the elements of the domain model that implements the *Composite* design pattern [8] to build a complex hierarchy of glyphs. The UML diagram in Figure 4 shows this poly-glyph implemented through the Composite design pattern in Manifold.

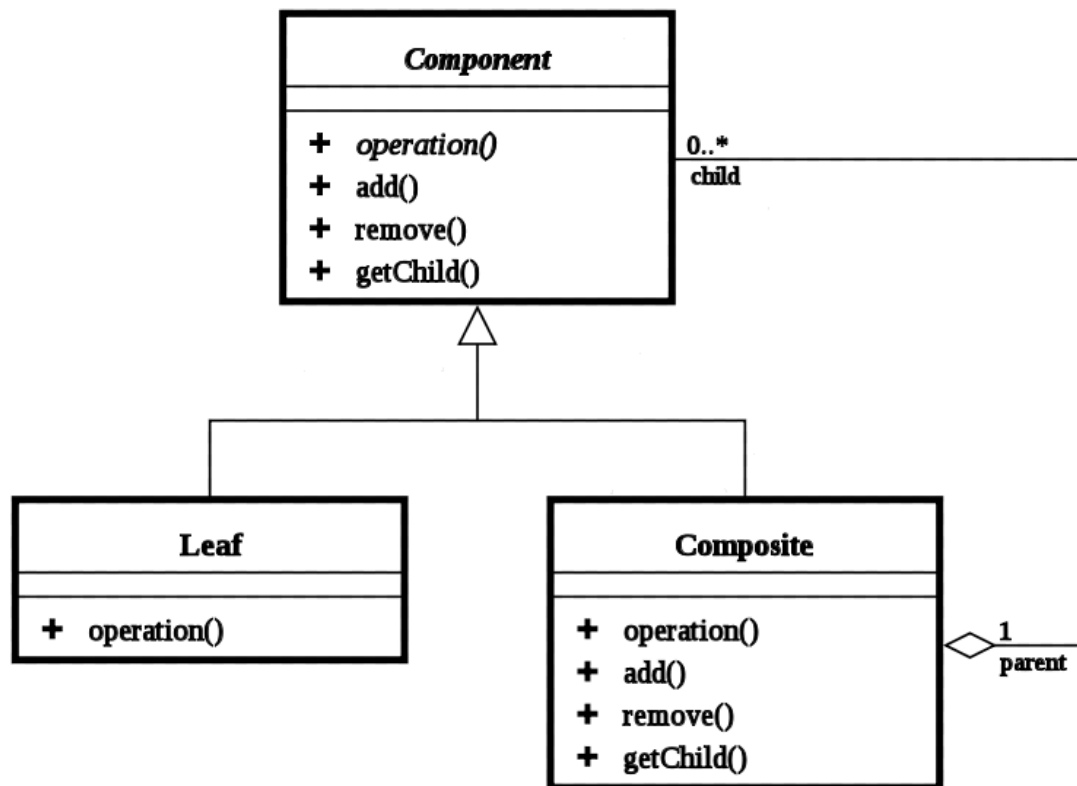


Figure 4: Composite Design Pattern

Table 1 summarizes the glyph inheritance hierarchy and their functionality in the current Manifold version.

Table 1: Manifold classes in the Glyph Inheritance hierarchy

Package	Class	Functionality
manifold	Glyph	An abstract base class for all the glyphs. Glyph is a visual representation corresponding to a model data element in a domain model. It visualizes the model's state changes.
manifold.impl2D	Glyph2D	An abstract base two-dimensional glyph implementation that

		implements some things common to both leaf (which have visual appearance, i.e., they can be rendered) and inner glyphs (a composite, i.e. a container for a group of glyphs).
<code>manifold.impl2D</code>	<code>GeometricFigure</code>	Leaf glyph represented via a simple geometric shape. This is the base class to be extended by the specific geometric figures. Notices that a leaf glyph has no children, so all composite-specific methods are defunct in this class.
<code>manifold.impl2D</code>	<code>TransformGroup</code>	Non-leaf (inner) glyph implementation used in construction of the hierarchical scene graphs. This is the composite version of the base Glyph (sometimes also called <i>poly-glyph</i>).
<code>manifold.impl2D.glyphs</code>		Defines all the implemented glyphs. See Chapter 5, section 5.2 for a detailed explanation of all the implemented glyphs.

2.2.1 Manifold Glyph Design

There are a few noteworthy properties of Manifold glyph's that is worth discussing here. These properties provide the semantics of how the glyphs are treated by the underlying domain model.

- Glyphs help the Tools/Manipulators (discussed in Chapter 5, section 5.1) to construct the manipulation event frames, to be sent to the domain. This includes simulation of interactive behaviors.

- They help in managing the structured graphics
- Glyphs are basically *hollow*, without any state. Their actual state is defined by the corresponding objects in the application domain. Glyph only mirrors what the application domain object notifies it.
- Glyphs cache their state information to improve performance. This is especially needed if the domain is located across the network. The look-up table represents the glyph's attributes as a set of $\langle \text{property}, \text{value} \rangle$ pairs.
- Glyphs provide shadowing to other glyphs. They are not “real” in the sense that the rest of the framework simply does not know about them. They are normally *structurally invisible*, meaning that they cannot be addressed and messaged to. They can be used to perform input handling or filtering, to decorate figures with shadows, borders, or bevels, and to perform layout alignment such as centering. Examples are *Highlighter* (`manifold.impl2D.glyphs.Highlighter`), *Connectors* (`manifold.impl2D.glyphs.Connectors`).

2.3 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. A Glyph listens (indirectly, via the parent Viewer) for changes of its underlying model. Upon receiving a change notification, it re-computes its own appearance, but it cannot redisplay itself because the actual display depends on how this glyph relates to others. An external class

must redraw other glyphs affected/damaged by this glyph's changed appearance. This is the role for the `Viewer` (package `manifold`). `Viewer` combines the View-Controller parts of the Model-View-Controller design pattern. All the communication between the View-Controller and Model is channeled through the `Viewer`.

The underlying GUI environment is specified by Java Swing, and captures the events from input device(s) and delivers them to the `Viewer`, since they happen within the viewer's window. The `Viewer` must then determine to which glyph(s) the event is directed to and dispatch the event to the target glyph(s). It is a good idea to keep the viewer implementation independent of the input device(s). We do this by `Viewer2DImpl` and `ViewerMouseListener` (package `manifold.swing`).

The frame rate is controlled by the class `Display` (package `manifold`) that runs in a separate thread. Extreme care should be taken on how the frame rate should be controlled. Answering each glyphs redraw request individually can be a resource consuming process (in terms of extra threads and memory). `Manifold` solves this by recording individual requests, but the actual redraw takes place only when the `Display` invokes `Viewer.redraw()` which in turn is controlled by the `Controller` (package `manifold`), standing as the gateway between the domain and the presentation.

2.4 Controller

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.

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. This parsing of input device events is performed by the Manipulator (package manifold) abstract class to which is implemented as an inner class of different Manifold tools (encapsulating the semantics of user interaction with application) to define the grasp, manipulate, effect cycle for the particular tool.

2.4.1 Manipulator

Manipulator encapsulates a Tool's manipulation behavior and is responsible for providing visual feedback during a manipulation sequence. Typical visual feedback is achieved by redrawing a rubber band using the XOR technique. A new Manipulator object is instantiated the moment the user starts a new interaction cycle and is 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.

A Manipulator generates *Frames*, 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 [64].

Most of these user-generated events are directed at the glyphs shown in the workspace, i.e., the viewer associated with the tool that created this manipulator. When an input event occurs, manipulator specialized in glyph manipulation has to first determine the glyph to which the event is actually directed to. The process of finding the relevant glyph is performed by the class `manifold.Traversal` (`pick()` operation) of the scene graph contained in the associated viewer.

2.4.2 Controller Communication

The controller implementation must specify a well-known list of the verbs that will be used in the event frames (see `EventFrame` package `manifold`) 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 knows what action(s) to take in response to particular event frames. Of course, there is no need for manipulators to know neither what those actions are nor what their meaning is.

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";
```


In short, 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.

Chapter 3

Developing Backend Applications

A User Interface (UI) is generally tightly coupled to the business logic of the application, and a great deal of modification is required to model it to separate the application logic. Manifold is an answer to such an issue. It offers a generic user interface framework where different application designers could develop separate application logic using the same Manifold front-end. The only need being the presence of a proper communication channel between Manifold and the backend applications (the domain model).

3.1 Introduction

In this chapter I will be emphasizing on the generic nature of Manifold, and how I used it to create a separate example backend application fronted by the Manifold interface that constituted a major portion of my thesis. This chapter provides a practical introduction to developing applications on Manifold by elaborating the framework for constructing a backend application. I will be considering the area of *Pattern Recognition* [46]. It must be noted that this does not represent a facility provided with Manifold, but how different application programmers could come up with different semantics to develop powerful applications and then visualize them on Manifold using a set of communication protocols. Also, this application does not appear with the Manifold distribution package. Manifold simply provides a visual feedback to the user by initiating

a separate backend application to perform semantic processing of data, where the semantics are application specific. In summary, applicability of direct manipulation does not depend on the spatial nature of the underlying data. All that matters is that the developer can come up with a spatial *representation* of the domain.

3.2 Communication Protocols

Consider a scenario, where you have an electric system, where in you use a manual switch to operate the electric devices and turn them on/off. Now, you don't always want to manually operate the switch, rather it would be highly convenient if you could just pass in a trigger to operate the switch from your computer, sitting in the comfort of your room. This could be done by developing your own system including the user interface and application logic to manipulate the switch; however it would be a time and resource consuming process. Thus, it would be a nice idea just to define your application logic and *attach* it directly to a direct manipulation device that could be used to visualize your application semantics. Manifold as a generic user interface provides this capability.

In order to make any backend application interact with the Manifold UI, a set of protocols have been defined. These protocols were also used by us in developing applications for the Manifold framework itself viz. *text* and *speech* recognition which provided it multimodal capabilities, as will be discussed in Chapter 4, section 4.3. This section would discuss such protocols necessary to begin developing applications that could run on Manifold front-end. If you are interested in developing applications or using

your previously developed applications that could be visualized on the Manifold UI, this section would be highly beneficial. The key idea here is how information exchange should occur between Manifold and backend applications without the need to modify the business logic of Manifold or the application. Figure 5 shows high level information exchange between Manifold user interface and other applications over a set of communication protocols. The next few sections would discuss these protocols in detail.

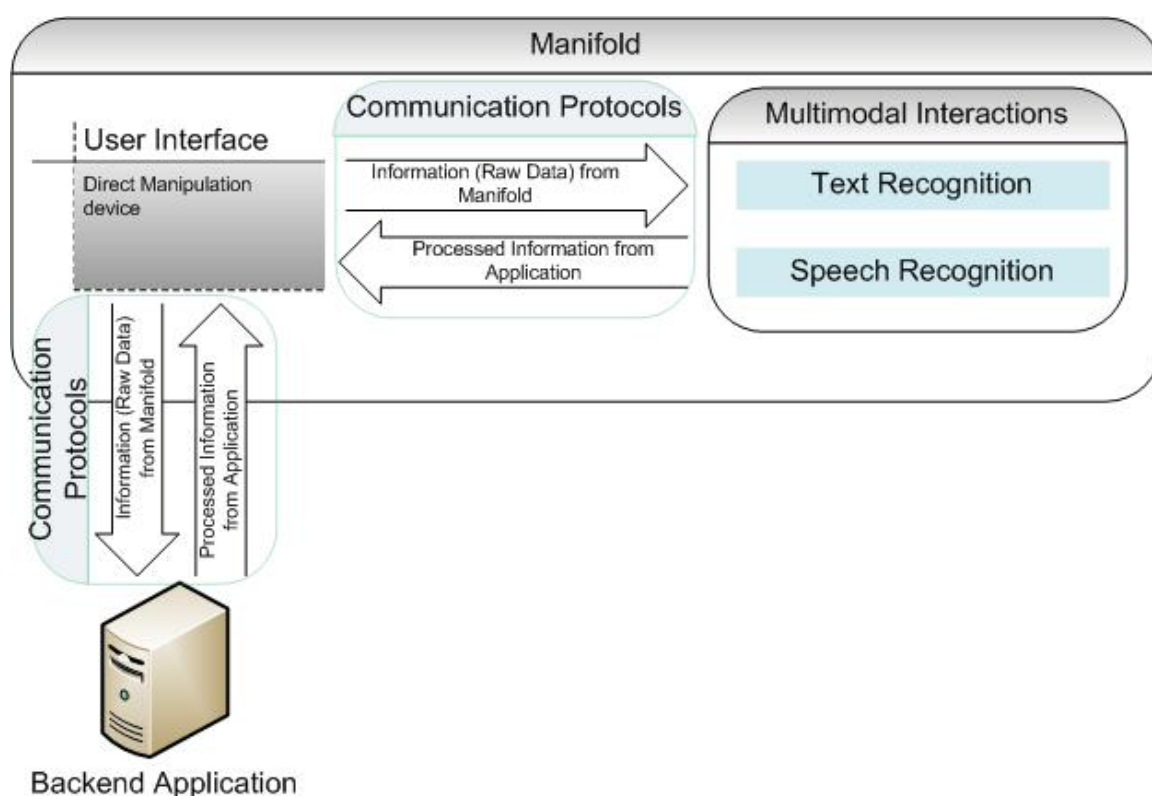


Figure 5: Manifold UI Interaction with other applications. Notice same protocols are used for interacting with multimodal interaction techniques as for other backend applications

3.2.1 Information Exchange: Formats

The first and the foremost requirement for developing backend applications that could run on Manifold frontend is that there should be a level of understanding between

them on the grounds of data format being interchanged. Manifold's *controller* acts as a single gateway between the presentation and domain modules of the system. It takes input from the user and instructs the *model* and *view* to perform actions based on that input. These end user actions could be transferred to the backend application logic in one of the following formats:

- Simple plain text input (as used by the *Text Recognition* application, Chapter 4, section 4.3.1).
- Manifold *EventFrame* (property-value pairs) constructed from the event-frame interpretation (these could be transferred by using a glyph's cached state, used by Pattern Recognition, as will be discussed in section 3.3).
- Speech input to the application (as used by the speech recognition application, Chapter 4, section 4.3.2)

By using these inputs, the application can perform any semantic processing on the input raw data and provide Manifold back with the information to be visualized on its viewer. The main requirement is that the information being transferred back to Manifold must be in the format that Manifold could understand and the *controller* could consequently instruct the *viewer* to perform the necessary action. Thus the information being transferred back to Manifold must be in form of event-frame (property-value pairs), which it understands and can parse easily without any modification. This information can be in the following formats:

- A JSON [50] object which forms an unordered collection of name/value pairs. Its external form is a string wrapped in curly braces with colons between the names and values, and commas between the values and names. This is used by the

Pattern Recognition application (section 3.3). It is helpful in representing hierarchical data, and can be easily interpreted on Manifold client (as Java provides interpretation libraries for JSON).

- A comma separated string to hold the event-frame (property-value slot) in a *<key, value>* respectively. This is used by the Text Recognition application (Chapter 4, section 4.3.1). This is mainly helpful if you are dealing with non- hierarchical data.

A Manifold EventFrame mainly comprises of following parts:

- Action verb (ADD/DELETE node)
- Node properties (node id, node type)
- Glyph's graphical attributes (color, width)

An example *EventFrame* can be:

```
{verb=setProperties, nodeType=rectangle, nodeId=node-4, line.width=5.0,
line.color=java.awt.Color[r=0,g=0,b=0]}
```

3.2.2 Information Exchange: Medium

Having discussed the formats of information exchange, now we will discuss the medium that carries this information over a communication channel between separate servers holding Manifold and backend applications.

The communication mediums that could be used for this purpose are:

- SOAP [13]: Simple Object Access Protocol is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks, relying on XML [12] for its message format. The web service

is responsible for taking in input from Manifold, passing it to a remote application server, and returning back the result in the form of comma separated string property-value pairs that could be understood by the *controller* to transfer the necessary actions to the viewer. This will be used for the purpose of text recognition for the generation of custom glyphs (Chapter 6, section 6.6.3).

- JSON-RPC [87]: A remote procedure call encoded in JSON, allowing bi-directional communication between the service and the client. A JSON invocation can be carried on an HTTP request where the content-type is application/json. This process is mainly used by the Pattern Recognition application, as will be discussed in detail in the next section.

Using the above methodologies for information format and transfer, backend applications could be attached with Manifold, accepting and passing information to the *controller* that can interpret and visualize the user input and domain model information on the *viewer*.

3.3 Example Application: Pattern Recognition

Manifold is a graphical user interface that allows drawing graphics (glyphs) on its viewer. It could be used by a large variety of users, who could use it to draw glyphs that are specific to their area of interest. An electrical engineer can use multiple glyphs to draw an electrical circuit. For example, a “simpleSwitch” may consist of a bulb (an ellipse type glyph), two boxes (rectangle type glyphs) i.e. circuit elements, and a wire (line type glyph) connecting the two boxes. When the wire joins the circuit elements the

circuit is complete and the bulb lights up. These semantics may not form any meaning for Manifold, however for the user they may be important and should be dealt with accordingly. The user might want the bulb to light automatically (become yellow) if the wire connects the two boxes i.e. when the circuit is complete. For this purpose there must be some logic developed that could identify the current pattern of the glyphs present on the Manifold viewer and instruct Manifold to perform some action in return (make the ellipse glyph yellow).

This idea can be accomplished through the process of Pattern Recognition. According to [41], *Pattern Recognition* is "the act of taking in raw data and taking an action based on the category of the pattern". The following sections discuss the process of Pattern recognition and how it was developed for recognizing simple glyph patterns on Manifold, using separate application logic.

3.3.1 Introduction: Pattern Recognition

In order to provide a better Human Computer Interaction mechanism it is necessary to build highly intelligent machine that can do things like human beings. The motivation for this comes from the practical need to find more efficient ways to accomplish intellectual tasks that may include realization, evaluation and interpretation of information that may come from various raw data sources. The ability to perceive information and process it is intrinsic to human beings. However, developing algorithms to explain the intrinsic mechanisms of perception and to exploit its mathematical aspects is a difficult task.

The aim of Pattern Recognition is to classify patterns of data, either using knowledge already gained or by statistical information within the pattern. Pattern recognition is a very important field of computer science and extends its uses to areas such as healthcare, security, etc. It deals with mathematical and technical aspects of classifying different objects through their observable information, such as closeness of two images.

Conventional pattern recognition systems have mainly two components viz. feature analysis and pattern classification. *Feature analysis* includes the steps of parameter extraction and feature extraction. *Parameter extraction* step includes extracting relevant information for pattern classification from the input data in the form of a parameter vector. *Feature extraction* is a special form of dimensionality reduction to provide relevant information from the input data, where the parameter vector is transformed into a feature vector having a reduced representation instead of a full size input. Figure 6 shows the data flow during the process of pattern recognition.

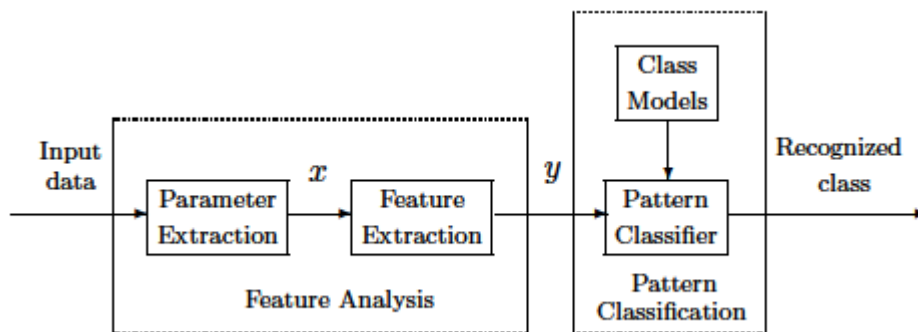


Figure 6: A conventional pattern recognition system

The development of the pattern recognition system, for identifying patterns generated on the Manifold viewer, mainly consists of above discussed steps, with a little modification and implementation differences (mainly for performance purposes), as will be discussed in the next sections. [45, 46] explains Pattern Recognition in detail.

3.3.2 Design: Pattern Recognition

The design of pattern recognition model for Manifold comprises of a relational database model. Figure 7 shows the schema with respect to the tables used in pattern recognition.

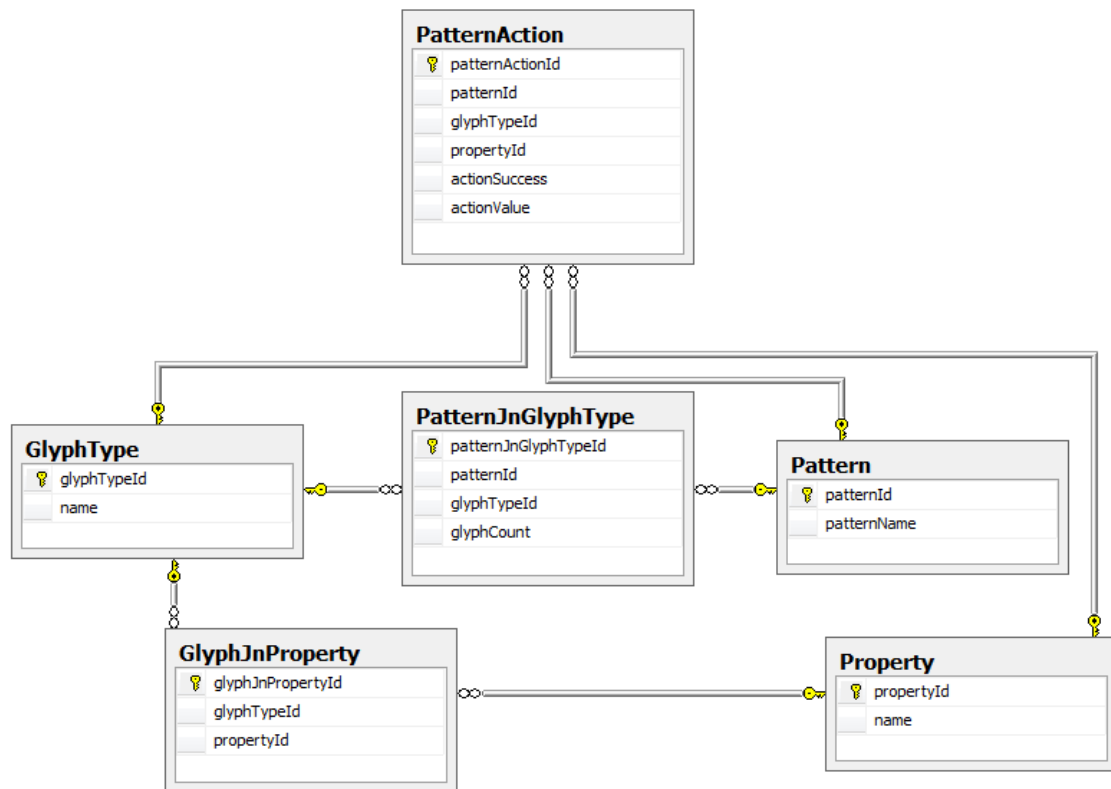


Figure 7: Database diagram for Manifold Pattern Recognition

The schema contains the following tables:

- **GlyphType:** Stores a list of Manifold glyph types viz. Rectangle, Ellipse, Line, Image, Text, etc.
- **Property:** Stores the name of properties that can be supported by Manifold glyphs.
- **GlyphJnProperty:** Specifies a many-to-many relationship between a glyph type and property (as different glyph types can have same properties).
- **Pattern:** Stores various pattern names (classes) like “simple switch”, “battery bulb” that could be identified by the underlying recognizer.
- **PatternJnGlyphType:** Specifies a many-to-many relationship between Pattern and a GlyphType, with the corresponding counts that a pattern can have for a particular glyph type (that is multiple patterns can have multiple glyph types with multiple counts and vice versa). For e.g. a “simple switch” pattern contains two rectangles, one ellipse and one line.
- **PatternAction:** Provides a mapping between Pattern, a GlyphType, and glyph Property. It specifies the necessary action to be taken by the underlying application on a glyph’s property present in the pattern, in case the pattern match is successful or unsuccessful. For e.g. when a “simple switch” pattern is matched, the bulb (ellipse) should be lightened (fill color property should become yellow).

Having discussed the design of the pattern recognition model, it would become easier to understand the data flow in the Manifold Pattern recognition system.

3.3.3 Data Flow in Pattern Recognition System

Data flow in a typical pattern recognition system is shown in Figure 8.

The first stage includes processing the collected information of an object, $x(t)$ by a parameter extractor. Information relevant to pattern classification is extracted from $x(t)$ in the form of a p -dimensional parameter vector x . x is then transformed to a feature vector y by a feature extractor which reduces the dimensionality of the input data vector and makes the input data suitable for pattern classifier. The feature vector has a dimensionality of m ($m \leq p$). The feature vector is assigned to one of the K classes, $\Omega_1, \Omega_2, \Omega_3, \dots, \Omega_k$, by the classifier based on a certain type of classification criteria.

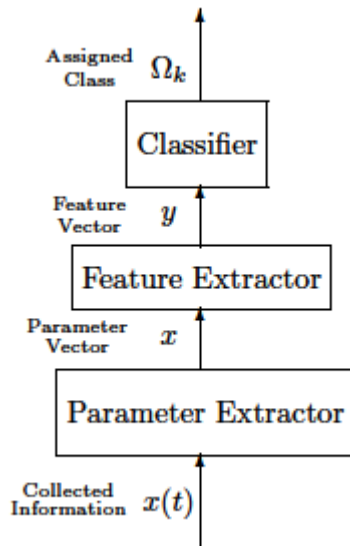


Figure 8: Data flow in pattern recognition system

In this thesis, I extend the process of classification further to exact pattern matching, where after a pattern class has been identified by the recognizer, restrictions are imposed on the class, to match it with certain criteria for that particular class, and take

the necessary action if those criteria are matched. Figure 9 outlines a higher level data flow between Manifold and the Pattern recognition application.

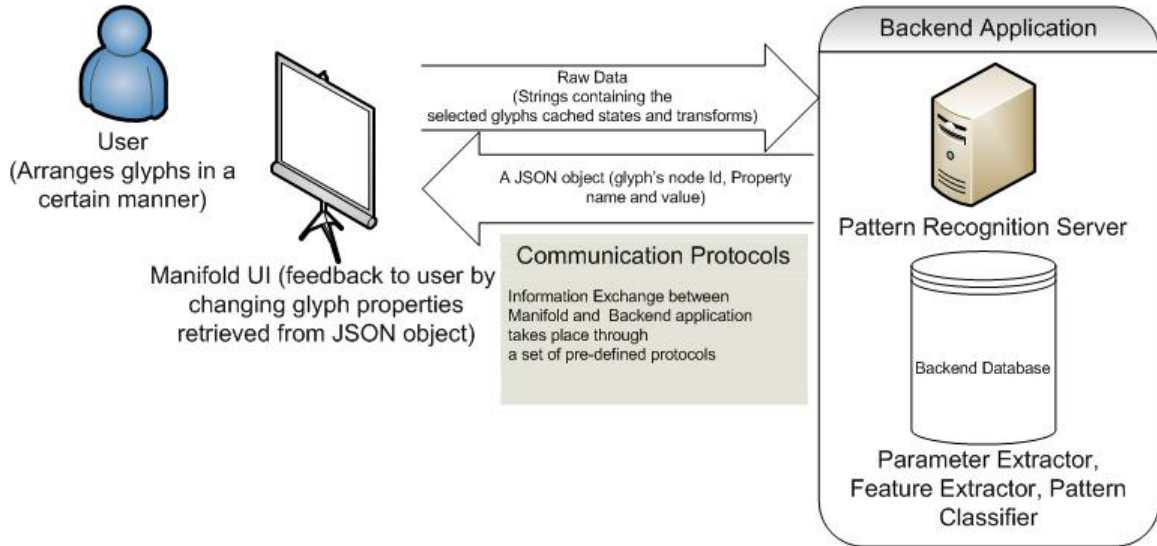


Figure 9: Higher Level data flow between Manifold and Pattern recognition system

The next four sub-sections will discuss the above outlined process in context with Manifold.

3.3.3.1 Unknown Object for Pattern Recognition System

The process of pattern recognition starts with gathering the unknown raw data and passing it to the recognition system. This raw data, in the current context, comes from the Manifold application. Manifold as a graphical editor understand the terminology of glyphs, their properties and values as specified by the `EventFrame` (package `manifold`) and understood by the underlying model. When we draw glyphs on Manifold, they belong to classes like rectangle, line, ellipse, etc., which independently do not provide any meaningful information. However, if these glyphs are arranged in certain

pattern, they could be used to provide certain visual feedback to the user, depending on the pattern. For example, when these glyphs are arranged so that they form a simple switch circuit, as interpreted by the user, to Manifold they are still glyphs. Thus, in order to provide visual feedback to the user, Manifold would initiate a separate backend application to perform semantic processing of data, where the semantics are application specific. This back-end logic would provide these patterns form meaningful representation on the viewer.

A user could select multiple glyphs using the Manifold's Selector (package `manifold.impl2D.tools`) tool, by drawing a rubber-band object around them. It creates a `SelectorManipulator` (refer to Chapter 5, section 5.1 for more details on tools and manipulators). In the method `effect()` of the `SelectorManipulator`, the selected glyphs are passed to a function called `glyphPositionRecogniser(Glyph[])`. This method gets the cached state and the transform matrix of all the selected glyphs, which forms the raw data (obtained using object serialization in Java (TM)), and passes it to the Pattern Recognizer (that resides on a separate server) over the communication protocols, in form of text. Example of certain cached state raw data (information pertaining to different glyphs separated by pipes i.e. '|') could be:

```
@cachedState = '{line.width=5.0, nodeType=rectangle,
transform=[D@8edb84,
source=manifold.impl2D.tools.Creator$CreatorManipulator@edf1de,
line.color=java.awt.Color[r=0,g=0,b=0], verb=setProperties,
nodeId=node-4} |{line.width=5.0, nodeType=rectangle,
transform=[D@1a2264c,
source=manifold.impl2D.tools.Creator$CreatorManipulator@804a77,
line.color=java.awt.Color[r=0,g=0,b=0], verb=setProperties,
nodeId=node-5} |{line.width=5.0, nodeType=line, transform=[D@1202f4d,
source=manifold.impl2D.tools.Selector$SelectorManipulator@196f8,
line.color=java.awt.Color[r=0,g=0,b=0], verb=setProperties,
```

```

nodeId=node-6} |{line.width=5.0, nodeType=ellipse,
fill.color=java.awt.Color[r=255,g=0,b=0], transform=[D@b749a5,
source=manifold.impl2D.tools.Selector$SelectorManipulator@5e832b,
line.color=java.awt.Color[r=0,g=0,b=0], verb=setProperties,
nodeId=node-7}'
,@transform = 'tx=800.5, ty=302.5, theta=0.0, xs=83.0, ys=41.0
|tx=1014.5, ty=300.0, theta=0.0, xs=71.0, ys=36.0
|tx=912.5000000000001, ty=295.0, theta=0.0, xs=191.00000000000003,
ys=8.526512829120879E-14 |tx=903.5, ty=149.0, theta=0.0, xs=61.0,
ys=40.0'

```

These glyphs form the unknown object and the text utterance containing their cached states and transform matrix, forms the raw data for the recognizer. It should be noted here that no modifications were done to Manifold; the existing information was gathered and passed on to separate server side logic. Figure 10 shows a “simple switch” circuit that shows how these unknown objects looked on Manifold.

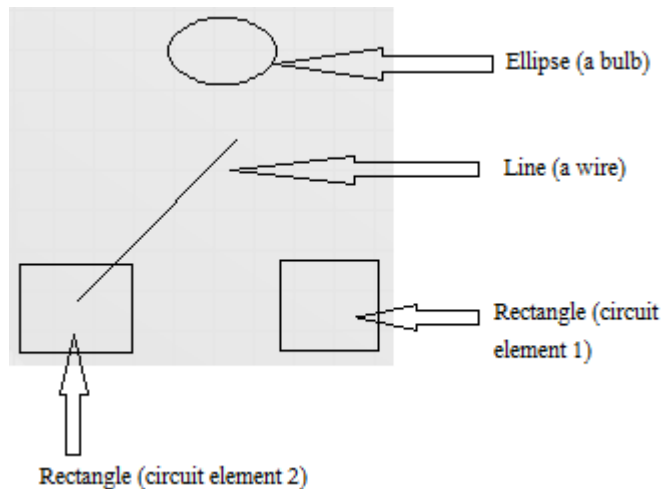


Figure 10: User trying to visualize a pattern via a number of glyphs (unknown object for Pattern Recognition system) on Manifold.

After the raw data has been collected, it is passed on to the Feature Analysis stage, where Parameter and Feature extraction takes place.

3.3.3.2 Feature Analysis: Parameter Extraction

A glyph's feature in Manifold is specified by the property-value pairs of its various editable attributes. The raw data collected in form of the cached state and transform text utterances needs to be parsed and tokenized so that important features could be extracted from it. The first step in this process is parameter extraction. The process of *parameter extraction* is used to extract important information from the input data in the form of a *p-dimensional* parameter vector x . This process extracts the information related to various glyphs from the raw data. The raw data, from the previous section, when passed on to this stage, generates the parameter vector x .

The figures (Figure 11, Figure 12, Figure 13 and Figure 14) show the parameter vectors generated as a result of the parameter extraction stage. These parameters contain the property-values pairs of a glyphs editable attributes (color, type, etc.) and its transform (width, height, rotation and location), as evident from these figures. However, it should be noted here that the dimensionality of these parameter vectors is very high and needs to be reduced for the sake of less computational cost and system complexity. Due to these reasons we employ the important step of feature extraction that outputs a feature vector (containing the geometric coordinates of a glyph that can be used to perform any kind of geometric computations) as explained next.

ItemName	ItemValue	tName	tValue
line.width	5.0	tx	800.5
nodeType	rectangle	ty	302.5
line.color	java.awt.Color[r=0,g=0,b=0]	theta	0.0
nodeId	node-4	xs	83.0
		ys	41.0

Figure 11: Parameter $x(1)$

ItemName	ItemValue	tName	tValue
line.width	5.0	tx	1014.5
nodeType	rectangle	ty	300.0
line.color	java.aw...	theta	0.0
nodeId	node-5	xs	71.0
		ys	36.0

Figure 12: Parameter $x(2)$

ItemName	ItemValue	tName	tValue
{line.width	5.0	tx	912.5000000000001
nodeType	line	ty	295.0
line.color	java.awt.Color[r=0,g=0,b=0]	theta	0.0
nodeId	node-6	xs	191.00000000000003
		ys	8.526512829120879E-14

Figure 13: Parameter x(3)

ItemName	ItemValue	tName	tValue
{line.width	5.0	tx	903.5
nodeType	ellipse	ty	149.0
fill.color	java.awt.Color[r=255,g=0,b=0]	theta	0.0
line.color	java.awt.Color[r=0,g=0,b=0]	xs	61.0
nodeId	node-7	ys	40.0

Figure 14: Parameter x(4)

3.3.3.3 Feature Analysis: Feature Extraction

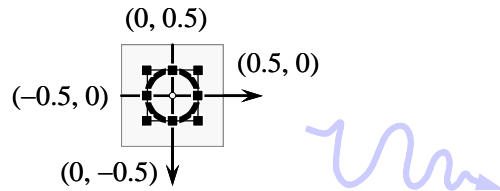
Feature extraction step does transformation on the parameter vector x to a feature vector y , which has a dimensionality ($m \leq p$). I follow the process of independent feature extraction [47, 48], by projecting the original parameter vectors onto a new feature space through a linear transformation matrix.

$$x \text{ (Parameter vector)} \xrightarrow{T \text{ (Linear Transformation Matrix)}} y \text{ (Feature vector)}$$

The linear transformation matrix that we consider here, transforms the parameter values tx, ty, xs, and ys (representing the x-translation, y-translation, x-scale, and y-scale respectively of a glyph in the Manifold viewer space), to the four point geometric

coordinate system ($\{x_1, y_1\}, \{x_2, y_2\}, \{x_3, y_3\}, \{x_4, y_4\}$). Figure 15 shows a glyph's local coordinates and its corresponding transformation matrix as represented in Manifold. For further detail on the transformation matrix and Manifold's coordinate space, the reader is advised to refer [4], Chapter 4.

Glyph in its local coordinate system:

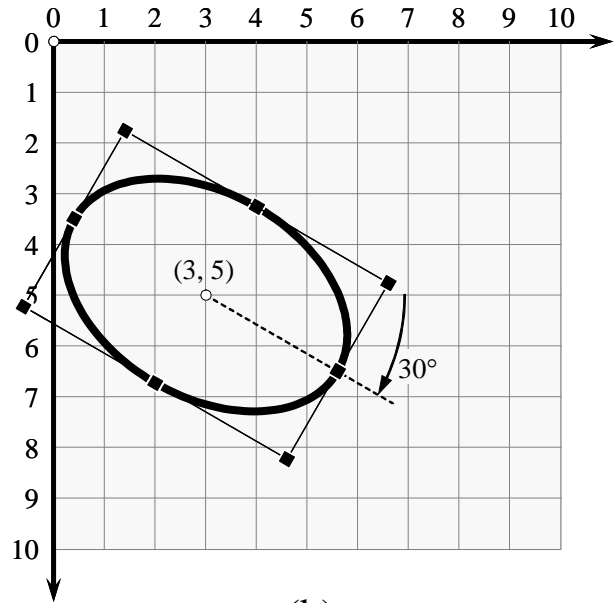


Transformation matrix:

$$[t_x, t_y, \theta, s_x, s_y] = [3, 5, 0.52, 6, 4]$$

$$\pi/6$$

(a)



(b)

Figure 15: (a) Glyph's prototype as represented in its local coordinate system. (b) Glyph transformed in the global coordinate system: positioned at (3, 5), width scaled to 6 and height to 4, and rotated by $\theta = 30^\circ = \pi/6$.

The resulting feature vector that contains the geometric coordinates of a glyph can be used to perform any kind of geometric computations, such as finding intersection points, checking whether a point lies in a plane, points of intersection of two glyphs, etc. This would allow the pattern recognition application, separated from Manifold, to compute the relative positions of glyphs with respect to each other as well their position on Manifold workspace, which makes pattern classification and similarity matching

possible, as discussed next in section 3.3.3.4. Figure 16 shows the feature vector y generated from the parameter vector x , as obtained in previous section.

GlyphNodeId	GlyphName	GlyphType	line.width	line.color	line.stroke	fill.color	tx	ty	th...	xs	ys	x1	y1	x2	y2	x3	y3	x4	y4
node-4	tempGlyph	rectangle	5.0	java.aw...	NULL	NULL	800.5	302.5	0.0	83.0	41.0	759	282	842	282	842	323	759	323
node-5	tempGlyph	rectangle	5.0	java.aw...	NULL	NULL	101...	300.0	0.0	71.0	36.0	979	282	1050	282	1050	318	979	318
node-6	tempGlyph	line	5.0	java.aw...	NULL	NULL	912...	295.0	0.0	19...	8....	817	295	1008	295	1008	295	817	295
node-7	tempGlyph	ellipse	5.0	java.aw...	NULL	java....	903.5	149.0	0.0	61.0	40.0	873	129	934	129	934	169	873	169

Figure 16: Feature vector y (with dimensionality $m \leq p$)

After generating the feature vector, the next step is classification.

3.3.3.4 Pattern Classification

The objective of pattern classification is to assign an input feature vector to one of K existing classes based on a classification measure. Classification measures include distance (Mahalanobis or Euclidean), likelihood and Bayesian *a posteriori* probability. The decision boundaries generated by these methods are generally linear and fall in the category of linear classification methods. Support Vector Machine (SVM) [49] on the other hand has a high computational flexibility and creates non-linear decision boundaries. Since, in our current implementation, all the calculations are performed with respect to determining the glyph transformations and 2-D geometric analysis that have low computational complexity, we will adopt the linear classification methods.

A **Pattern** is a quantitative or structural description of an object or some other entity of interest, arranged in the form of a feature vector as:

$$\begin{aligned} &[x_1] \\ \mathbf{X} &= [x_2] \\ &[x_n] \end{aligned}$$

where x_1, x_2, \dots, x_n are features. For example, a “simple switch” pattern may consist of a bulb, two boxes (circuit elements), and a wire connecting the circuit elements. When the wire joins the two boxes, the circuit is complete and the bulb lights up.

Class or **pattern class** is a set of patterns that share some common properties. The feature vectors of the same type of objects will naturally form one set. In the current context we will deal with only class of type electrical circuits, as they share common properties such as wires, battery, bulb, etc, and we can use Manifold glyphs to interpret them. Within this class we will use patterns like simple switch and battery bulb, to recognize and match them correctly.

A **classifier** creates a series of functions, apply them on the input vector, and output a value, based on which it assigns the input vector to one of the classes. In context to our current discussion on pattern recognition in Manifold, these functions could be measurement of Euclidean distance between two objects, and determining whether a point lies in a plane.

In the section 3.3.3.3 we discussed how the recognizer built the feature vector, from the raw data. The next step is determining whether the objects present in this feature vector, can be classified into a pattern or not. For this purpose, the database table **Pattern** is used which specifies the name of various patterns within the electrical circuit class. The relationship between various glyphs from the feature vector and this pattern name is developed through the table **PatternJnGlyphType** that specifies the objects (glyphs here) that should be present in a pattern and their corresponding count for that pattern. For example, a “simpleSwitch” pattern may consist of two rectangles (two circuit elements), a line (wire), and an ellipse (a bulb). If all the glyphs are present with their

respective count in the feature vector, we could say that there is a possibility of a particular pattern from the objects present in the feature vector, and a particular pattern has been classified.

The classification of a pattern doesn't necessary determine the completeness. For instance, a "simpleSwitch" pattern may be present in the feature vector, but may not be complete (the wire might not be connecting the two circuit elements or there might be some other circuit fault). For this purpose, we apply a series of geometric functions on the identified glyphs feature vector that determines their relative position on the Manifold viewer space using the geometric coordinates obtained in the feature vector. For the "simpleSwitch" pattern, the two rectangles must be connected by a line (to complete the circuit). On the basis of successful or unsuccessful pattern matching, a corresponding action is generated. This action is stored in the table `PatternAction` which provides the successful and unsuccessful actions, and the corresponding property name, and property value that should be send to Manifold. For example, if there is a success in the "simpleSwitch" pattern, the ellipse should be filled with color YELLOW and RED otherwise.

The final step is to assemble these key-value pair and pass them onto Manifold as a JSON [50] object, where it can be parsed easily, and the action could be rendered on the Manifold viewer space. An example JSON object for the above discussed pattern could be:

```
json
{matchedPattern:Pattern simpleSwitch matched successfully,success:1,actionNodeId:node-7,propertyName:fill.color,propertyValue:yellow}
```

Figure 17: JSON object send to Manifold over the communication channel

Table 2 shows the SQL stored procedures used in the process of Pattern Recognition along with their respective functionalities.

Table 2: SQL stored procedures for the process of pattern recognition

Stored Procedure Name	Function
RECOGNISER_GLYPH_POSITION	Main procedure that parses the input text utterance containing glyph cached states and transforms to generate parameter and feature vectors, and calls other procedures for pattern classification, matching and action generation.
RECOGNISER_CHECK_LINE_INETERSECTS	Checks whether two lines given their geometric coordinated intersect in a 2-D plane.
RECOGNISER_CHECK_POINT_IN_PLANE	Checks whether a given point lies in a 2-D plane.
RECOGNISER_GENERATE_PATTERN_ACTION	Generates a pattern action (property-value pair) depending on whether pattern recognition has been successful or not.

3.3.3.5 Rendering on Manifold

The JSON object contains all the necessary information that Manifold can understand and directly render the information to the *Viewer*. If the JSON object is null, or any of its properties are null that are required by Manifold, then no action occurs.

The code stub to parse the JSON object lies in the class `Selector.java` (package `manifold.impl2D.tools`):

```
try {
    JSONObject outer = new JSONObject(ji); // the outer objet from a
    //string input ji
    if (outer != null) {
        // Parse the name/value pairs
```

```

        success_ = outer.getString("success");
        matchedPattern_ = outer.getString("matchedPattern");
        propertyName = outer.getString("propertyName");
        propertyValue = outer.getString("propertyValue");
        actionNodeId = outer.getString("actionNodeId");

        System.out.println(matchedPattern_);

        if (actionNodeId != null || !actionNodeId.isEmpty()) {
            performAction(actionNodeId, propertyName,
                propertyValue);
        }
    }
} catch (Exception e) {
    System.out.println(e.toString());
}

```

The parsed values are passed to the method `performAction(String, String, String)` in the same class which sends the `EventFrame` to the controller to render the specific action on the viewer. The code stub for this is:

```

// 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.SOURCE, this);
slots_.put(EventFrame.NODE_ID, actionNodeId);
slots_.put(propertyName, pValue);
viewer.getController().sendAsyncEvent(
    new EventFrame(slots_));

```

Here, the backend application notifies the controller of the property change, and the corresponding node Id, property name, and property value, retrieved from JSON object are send to it, which are then rendered on the viewer. Figure 18 and Figure 19 shows the interaction and output on Manifold.

The process discussed above could be extended for more complicated pattern recognition and matching. The developer has to define his new semantics, and the above discussed architecture could be used for any kind of visual pattern recognition on

Manifold. This could become even more interesting if new glyphs of type wire, resistor, battery, etc. are defined on Manifold. The possibilities are endless, however, Manifold architecture is flexible enough to implement new glyphs and features and develop applications that could understand and manipulate these glyphs.

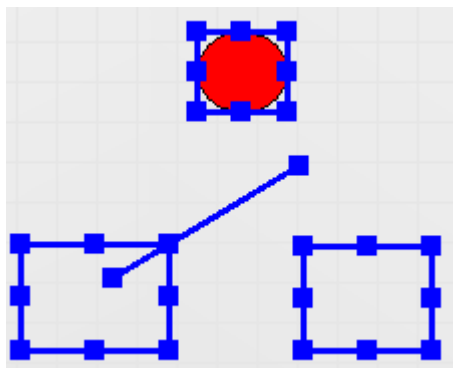


Figure 18: Pattern simple switch is classified, but not matched (red color of ellipse)

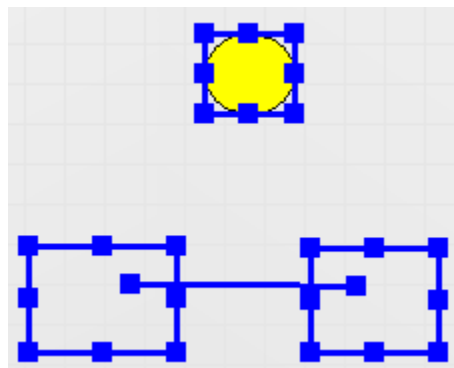


Figure 19: Circuit is complete (line connects two rectangles), and the pattern is matched (yellow color of ellipse)

To give an example of a more complicated pattern, a battery bulb pattern is shown in Figure 20 and Figure 21.

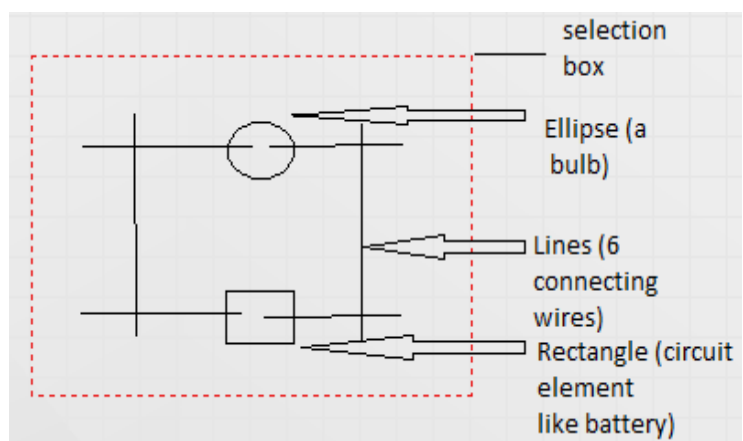


Figure 20: Visualization of a battery-bulb pattern on Manifold

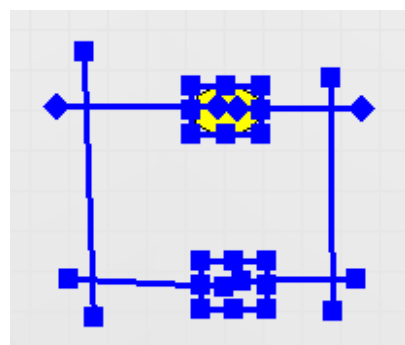


Figure 21: A battery bulb pattern classified and matched successfully (as all circuit elements are well connected).

Chapter 4

Enhancements and Multimodal Interaction Techniques

With the availability of a large number of user-interface tools, there was a need to add features in Manifold, and enhance the existing ones which were envisioned in the previous versions. These were discussed in brief in Chapter 1, section 1.3.2. There was also a need to provide user with multiple modes of interfacing with the Manifold framework making it a multimodal [6] user interface.

In this chapter I will be discussing some of the enhancements that were made to Manifold, and how they were carried out. I will also be discussing the areas of *Text Recognition* (also referred to as *Text Parsing*) [79], and *Speech Recognition* [42, 43] along with their implementation in the Manifold framework to make it an efficient interface for direct manipulation and multimodal interaction which plays an important role in HCI [1].

4.1 Introduction

Manifold is a GUI that operates on glyphs. In order to make its functioning fast and smooth, it was required to improve its functionality from different perspectives. This was done by eliminating some of the shortcomings from the previous versions of Manifold. Since Manifold is a work in progress, it was necessary to improve the functionalities related with the way how glyphs are selected, and their properties edited.

It was also necessary to provide user specific configuration of Manifold, so that different users could run different features, by including/removing features at run-time.

Also, in order to provide the users multiple ways of accomplishing a single task, and increase Manifold's capabilities of being a multimodal UI than just being a direct manipulation interface, new interaction techniques like text and speech were attached with Manifold. As discussed in Chapter 1, *Human Computer Interaction* plays an important role in designing any graphical editor or a user interface. As people start to use their PCs more, they start to identify ways in which they want to use peripherals for different tasks. Task specialization has led to a variety of keyboard and mice on the market; but, there are emerging more exotic interface technologies. We discuss the same in this chapter.

4.2 New Features and Enhancements

This section discusses some of the new features and enhancements that were added to provide a better user experience with the Manifold UI.

4.2.1 Editing Multiple Glyphs

The *Property Viewer* will be discussed in detail in Chapter 7, and how it uses Property Editors to edit different properties of different glyphs. Editing properties of multiple glyphs at the same time could be highly valuable to the user, especially in scenarios where different glyphs have to be provided with same properties. For e.g. the

user might want to fill different glyphs with the same color at once, which otherwise would require him/her “n” steps to change the *fill color* property of “n” glyphs.

In the previous versions of Manifold, multiple glyphs could be selected by drawing a selection box around them; however, their properties could not be edited, all at once. Figure 22 shows the Property Viewer on selection of multiple glyphs from previous version of Manifold.

To allow editing properties of multiple selected glyphs, all at once, some modifications were required to be made in the class `PropertiesViewer.java` (package `manifold.swing`). This class holds the panel containing all the property editors for the currently selected glyph, if any. Once a glyph is selected, the editor panel (`manifold.swing.PropertyEditorsPanel.java`) corresponding to this glyph type (with all *editable* properties is located in the lookup-table, a `HashMap`). The panel is added as a child component to this panel and displayed. If no glyph is selected, this panel is empty or (preferably) hidden.

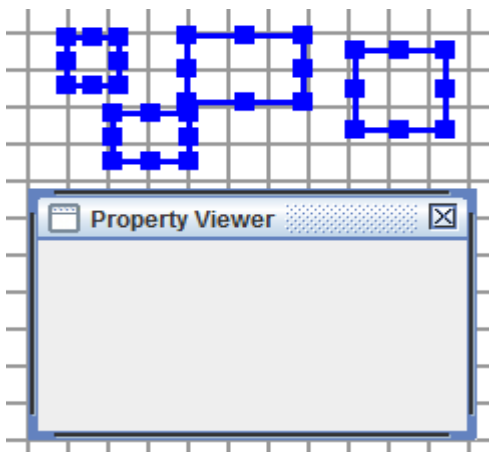


Figure 22: Empty Property Viewer on selection of multiple glyphs in previous versions of Manifold

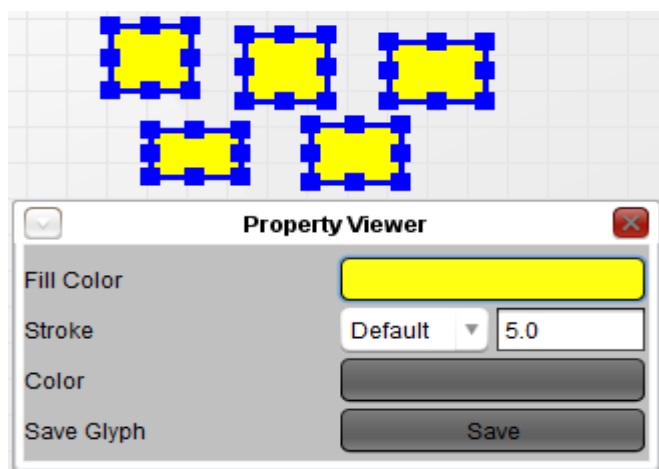


Figure 23: Editing properties of multiple glyphs in current Manifold version (Fill Color property being edited here)

In the previous version of Manifold, the identifier for the currently selected glyph was a `String` field. To make it accept multiple glyphs, it was changed to accept a `String` array of glyph `nodeId`'s. This identifier is set in the method `selectionsChange(SelectionsEvent)`, see `manifold.SelectionsListener.java`, which is called on a listener object to notify about changes to the list of selected glyphs in the viewer(s). The parameter specifies the event carrying information about the selections change. The method is modified such that it gets all the selected glyphs and then adds common *Property Editor*(s) to the *Property Viewer*, by looping through the selected glyphs and taking a union of the property-editors for all the selected glyphs. To avoid missing property-editors that were not common to two glyph types, we took the union of property-editors rather than intersection. For e.g. *Font Editor* is used only for Text glyph type, but *Color Editor* is used for both Text and Rectangle glyph types.

To support multiple glyphs' property editing, changes were also made to the classes in the package `manifold.swing.editors`. This was done so as to allow making event frames to allow editing properties of multiple glyphs and sending request to the application domain for the property change of all the glyphs. Following these modifications, it was possible to edit properties of multiple glyphs at once (Figure 23).

4.2.2 Poly Glyph

Handling multiple glyphs become even more important if we want to treat a selection of glyphs as a single object. This promotes the implementation of *Poly-Glyph*,

based on the Composite Design Pattern [8], feature in Manifold by altering the parent-child properties. The poly-glyph is implemented in class `TransformGroup` belonging to package `manifold.impl2D` and is created using the *Grouper* tool. The reader is advised to read Chapter 5, section 5.4.2 for an in-depth understanding of this feature.

4.2.3 Glyph Selection

In the previous version of Manifold, it was not possible to select glyphs of type “line” i.e. *Line* (`manifold.impl2D.glyphs.Line.java`) and *Link* (`manifold.impl2D.glyphs.Link.java`), by drawing a selection box around them. This behavior was due to the fact that the Selector tool was made to pick shapes of type `Rectangle2D` only (see method `gatherSelectionsAndCleanUp(InputDeviceEvent, Viewer)` in class `manifold.impl2D.tools.Selector.java`). As a result when a selection box was drawn around glyphs of type “line”, they were not picked as they had a bounding shape of type `Line2D`. This was done in method `getBoundingShape(boolean)` of class `Line.java`. The code stub from the previous version was:

```
public Shape getBoundingShape(boolean transformed_) {
    if (transformed_) {
        return shape;
    } else { // return the prototype
        return new Line2D.Double(
            GeometricFigure.POINT_1.getX(), 0.0d,
            GeometricFigure.POINT_2.getX(), 0.0d
        );
    }
}
```

To remove this behavior and allow selection of `Line2D` (line and linker) by drawing a selection box around them on the Manifold viewer, the bounding shape of the Line glyph was modified to return `Rectangle2D` as its bounding shape. The modified code stub is:

```
public Shape getBoundingShape(boolean transformed_) {
    if (transformed_) {
        return shape;
    } else { // return the prototype
        return new Rectangle2D.Double(
            GeometricFigure.POINT_1.getX(), GeometricFigure.POINT_1.getY(),
            GeometricFigure.POINT_2.getX() - GeometricFigure.POINT_1.getX(),
            GeometricFigure.POINT_2.getY() - GeometricFigure.POINT_1.getY()
        );
    }
}
```

By correcting the bounding shape, it was possible to select “line” glyph types by drawing selection box around them.

4.2.4 Manifold Configuration

Configuring the features of an application is very important from system resources/performance perspective. Hence it becomes essential in allowing configuration of Manifold before running it, by providing a *Wizard* [11] to select the necessary XML [12] files to configure it.

The Manifold configuration wizard is implemented in package `manifold.swing.wizard` (see [11] for example implementation), and allows selection of user configured/distributed XML files as per user requirement which enhances the application speed by limiting the functionality as per their needs. For e.g. the user could select custom configured XML files (“draw2D.xml”, “editors.xml”,

“glyphs.xml”, “menuItems.xml”, and “tools.xml”) as per their need. Figure 24, Figure 25, and Figure 26 shows the Manifold configuration wizard at various steps.

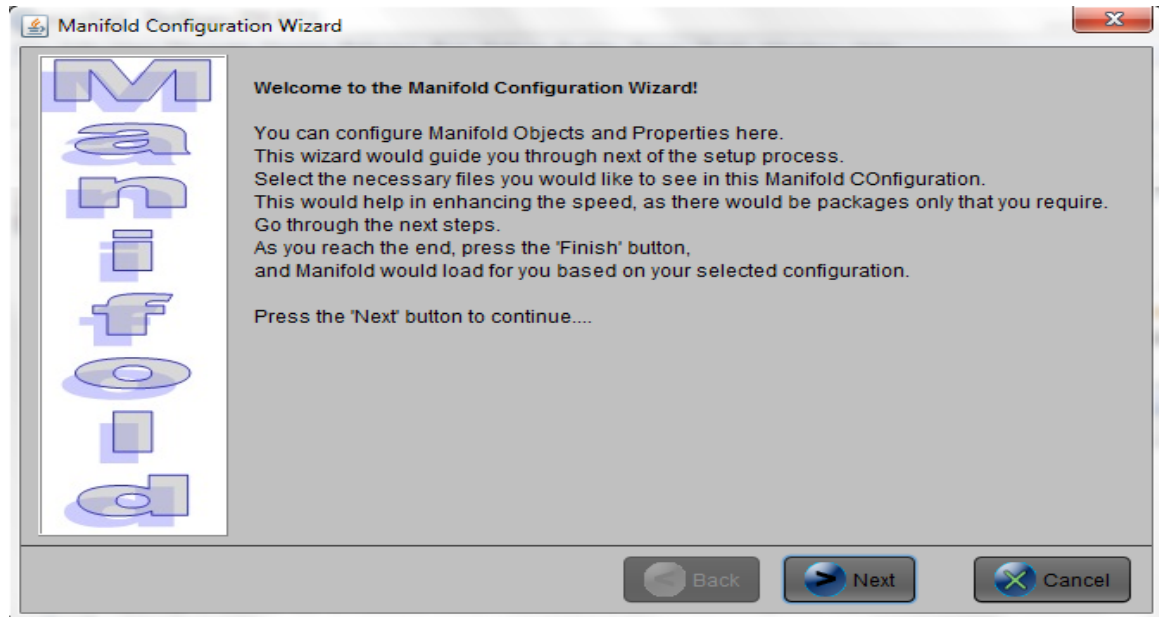


Figure 24: The Manifold Configuration Wizard, welcome text

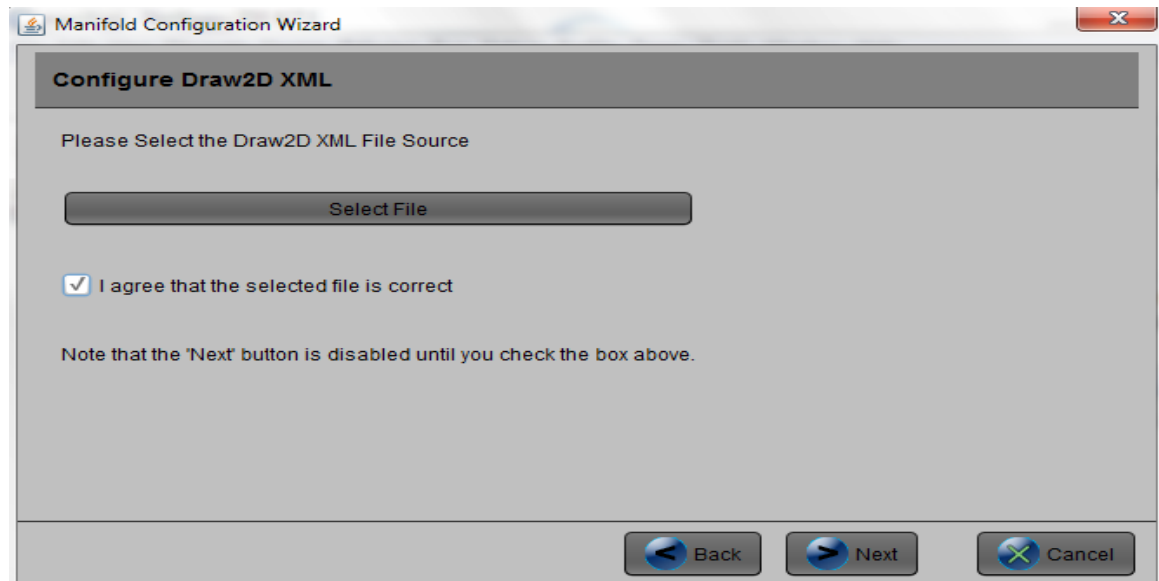


Figure 25: The Manifold Configuration Wizard, selecting an XML file

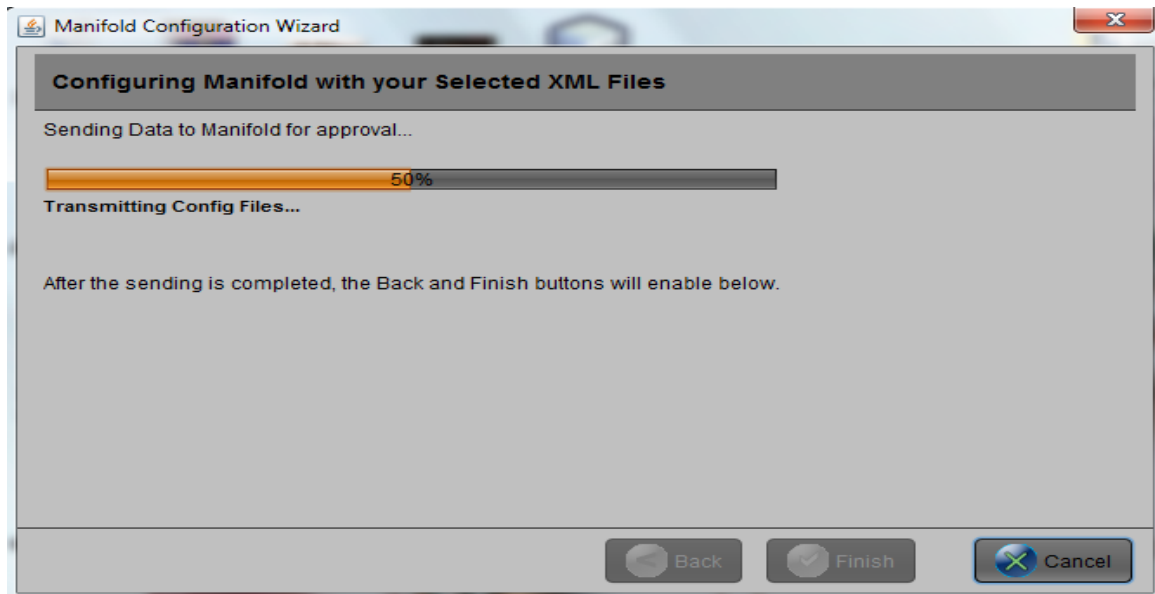


Figure 26: The Manifold Configuration Wizard, configuring application with selected XML files

4.2.5 Correcting Bounding Shapes

In the previous Manifold versions, the *bounding shape* and the highlighter (shadow glyph) of a glyph were misplaced with respect to each other. It was necessary that both of them coincide, as highlighter allows manipulation of underlying glyph by shadowing it. Hence, the bounding shapes for text and image glyph were corrected in the respective `draw()` methods of the classes `Text.java` and `Picture.java` (package `manifold.impl2D.glyphs`), to make the highlighter coincide with them.

4.3 Multimodal Interaction Techniques

This section discusses some of the new interaction techniques viz. speech and text that were created as separate applications and then attached (using predefined protocols) with the generic Manifold framework as discussed in Chapter 3, section 3.2.

4.3.1 Text Recognition

Text Recognition [79] may be defined as the process of parsing an input (a command) from the user and simplify it to something that the software can understand, to generate some visual (and non-visual) feedback. It is very intuitive to let a framework do work for you by just writing “what to do”, instead of manipulating different commands separately. Consider for example, if you want to draw a graphic on a user interface, the user interface provides you tools to create the graphic, change its appearance, position etc. In order to accomplish these steps, you have to provide various commands to the user interface, either by manipulating through mouse, or writing text commands (as in MS DOS). This is time consuming and manipulating all steps manually and making the graphics of correct shape, size, and at a certain location may be a painstaking process. Now, consider another scenario, where you can just type in what you want to do, all at once. For e.g. if you just type in “*Draw graphics X at location x, y with property Y*”, the user interface should be intelligent enough to understand the text, parse the text, consider into account synonyms of all words you typed in, and render the graphic on the interface with all the properties you provided (removing wrong values and considering only permissible values for that graphic object).

In this section I will be considering such a scenario, and how I utilized the generic nature of Manifold to render glyphs on its viewer, by parsing a simple (but meaningful) line of text. Manifold understands only properties and what values they can have for a certain glyph. Hence, if we provide certain input text to Manifold, separate server side logic has to be developed, in order to provide it property specific values that it can

understand (without any manipulation on Manifold itself). This nature of Manifold, would allow developers to develop different applications and attach them to its UI.

The need for the development of the process of Text Recognition will be discussed in Chapter 6, section 6.6.3, where the *Custom Glyph* menu item is described. The interface on Manifold and the communication with a separate application server were also discussed in the same section. The reader is advised to read the same to understand how the process interaction was carried on Manifold.

4.3.1.1 Design: Text Recognition

For the process of Text Recognition we present a database model that would allow recognizing a text pattern and consequently render graphics (glyphs) on the Manifold viewer.

Figure 27 shows the database model with specifications of the tables used for text recognition.

A general description of the tables is as follows:

- **Noise:** Stores all the noise words that can be possibly present in an input text utterance, and should be removed before manipulation. Example noise words could be “a”, “the”, etc. For more details on noise words, see [44].
- **Word:** Stores all the distinct words that have been passed into the input text, after removing the noise words. It stores a word only once i.e. if not already present.
- **GlyphType:** Stores a list of Manifold glyph types viz. Rectangle, Ellipse, Line, Image, Text, etc.

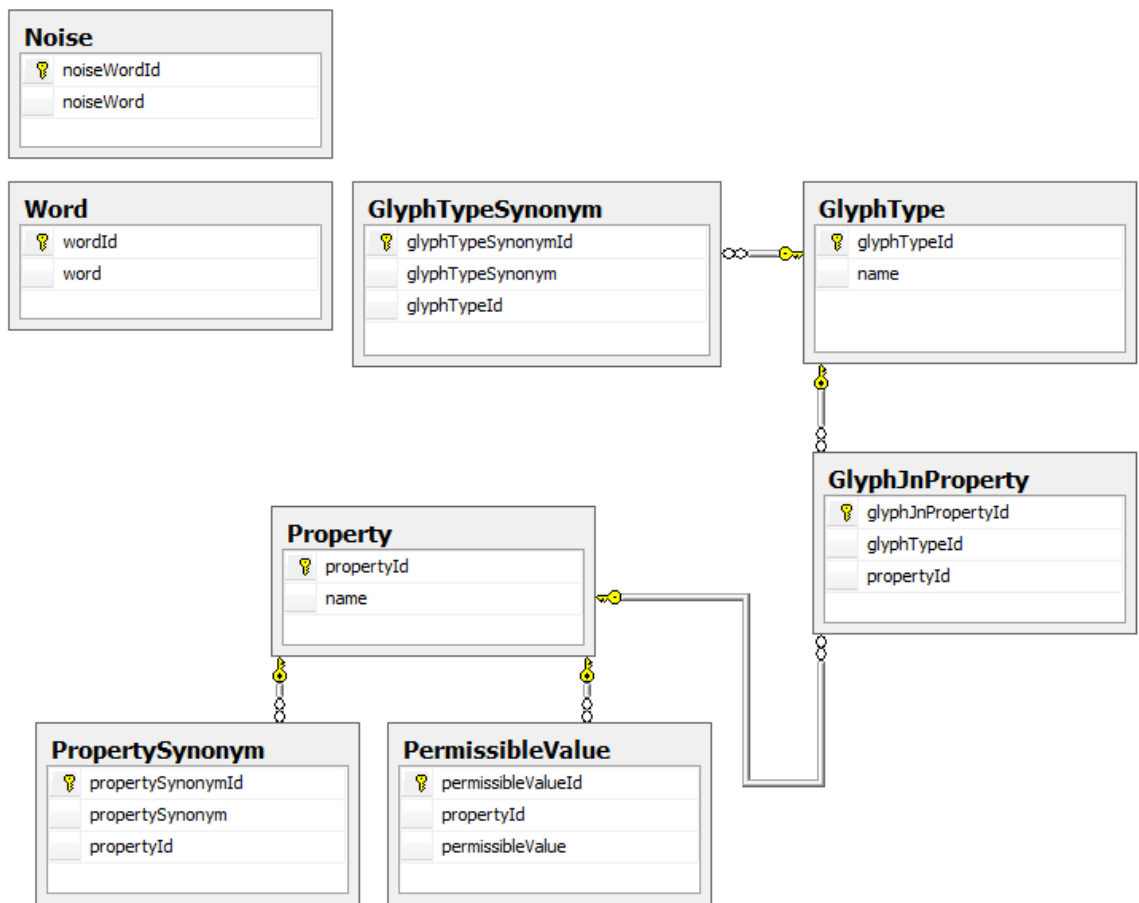


Figure 27: Database schema for Text Recognition

- **GlyphTypeSynonym**: Stores the synonym for glyph type names i.e. many-to-one relationship (a glyph type can be referred by many different names). For e.g. ellipse may be referred as oval, line may be referred as a segment, etc.
- **Property**: Stores the name of properties that can be supported by Manifold glyphs.
- **GlyphJnProperty**: Specifies a many-to-many relationship between a glyph type and property (as different glyphs can have same properties).

- **PropertySynonym:** Stores all the possible names by which a property can be called with (example width and breadth specifies the same property).
- **PermissibleValue:** Stores the type of values that are permissible for a particular property (for example `fill.color` can contain properties of type color; width, height should be of type numeric, etc.).

The next section describes the algorithm I built for the purpose of text recognition to meaningful values through parsing and cleaning, using the above model.

4.3.1.2 Implementation: Text Recognition

The flow chart in Figure 28 shows the basic steps in the process involved in Text recognition. The formal algorithm (pseudo code) is (the description follows):

Algorithm Text Recognition

Input: Text word utterance (inputUtterance)

Output: Glyph Profile (property-value pairs)

```

for each word in inputUtterance
    if the word is not a Noise word
        then Store the word and its position from the utterance in a Word-Position
            table
            Extract the glyph type (present as one of the words)

if the glyph type is not null
    then generate a Bigram Table from Word-Position Table (words with position
        difference = 1)

for each bigram in Bigram Table
    if a bigram forms a proper property-value pair (considering property-name
        synonyms and permissible values for the properties)
        then Store it as a part of the final Glyph Profile

return Glyph Profile

```

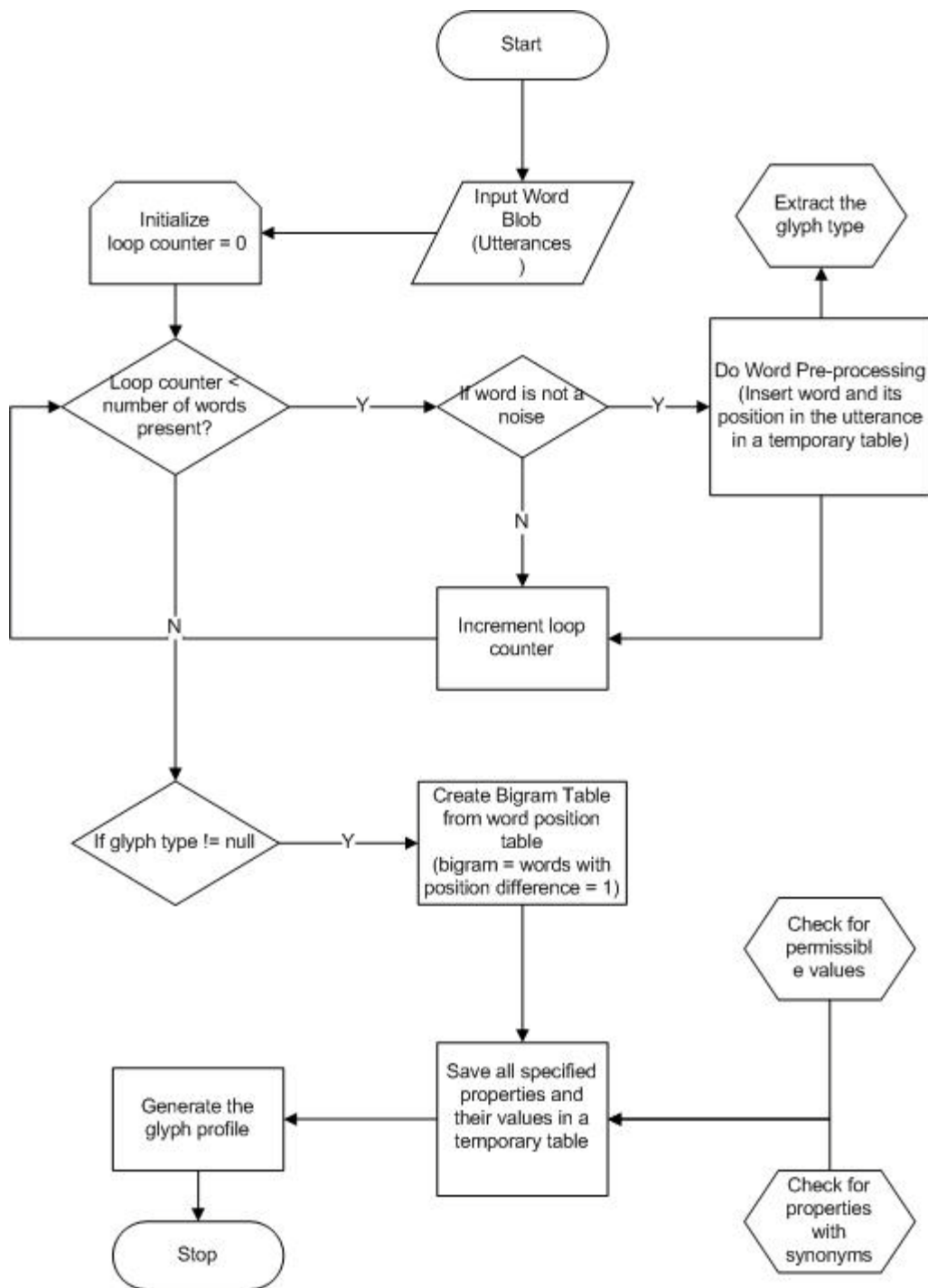


Figure 28: Flow chart depicting the entire process of Text recognition

When a user enters a word utterance (an input text string that contains actions that should be performed so that a particular glyph gets rendered on the Manifold viewer), it is passed on to separate server side logic through a SOAP web service as discussed in Chapter 6, section 6.6.3. The first step in the process is to do word preprocessing (*lexical analysis* [79]), to extract meaningful data (tokens) from the word utterance. For this purpose we start by cleaning the text, which is accomplished by splitting the sentence into words, removing the noise words by comparing them against the NOISE table and storing the words themselves in the Word table (only if they do not exist in the table). We then store the word (wordId of the stored word from the Word table) and its corresponding position in the text in a temporary table. For e.g. the preprocessed output for a word utterance “Draw a rectangle of width 45 and height 50 with fill_color red and rotation of 45 degree” is shown in Figure 29. Here all the noise words like “a”, “of”, “and”, and “with” are removed from the input utterance. The regular expression used here to split the input character stream in a set of meaningful symbols is defined by empty spaces so that we could obtain words from a sentence. It should be noted here that we also extract the glyph type name (or its synonym) in this process of lexical analysis only.

After data preprocessing, the next step is to retrieve the words with close proximity that are permissible and have meaningful information associated with them. Proximity here means two or more separately matching term occurrences are within a specified distance, where distance is the number of intermediate words or characters. By limiting the proximity to “two”, the property value pairs can be matched while the scattered words that are spread across the sentence can be avoided. For e.g. in the

sentence, “Draw a rectangle of width 45 and height 50 with fill_color red and rotation of 45 degree”, ‘width’ and ‘45’ occur next to each other as a property-value pair. We make use of bigrams (group of two words) for this purpose that help provide the conditional probability of a word given the preceding word, when the relation of the conditional probability is applied:

$$P(W_n / W_{n-1}) = P(W_{n-1}, W_n) \div P(W_{n-1})$$

That is, the probability $P()$ of a word W_n given the preceding word W_{n-1} is equal to the probability of their bigram, or the co-occurrence of the two words $P(W_{n-1}, W_n)$, divided by the probability of the preceding word.

This allows storing the preprocessed output in form of all possible bigrams formed (which is calculated using the word position difference of 1). The bigrams formed, by the output shown in Figure 29, are shown in Figure 30.

The next stage is *syntactic analysis* [79] that checks whether the bigrams form an allowable expression so as to extract the glyph specific properties and their values. This is done by comparing each row and corresponding column value against the glyph specific properties (rectangle here) from the Property table. We also check the possible property name synonyms by comparing them against the PropertySynonym table. After identifying a property, we check whether its value is permissible for the given glyph type by comparing the values against the table PermissibleValue. Finally, the glyph profile is generated containing the property value pairs that can be identified by Manifold. The glyph profile for the above input text is shown in Figure 31.

wordValuePosId	word	wordPosition
1	Draw	1
2	rectangle	2
3	width	3
4	45	4
5	height	5
6	50	6
7	fill_color	7
8	red	8
9	rotation	9
10	45	10

Figure 29: Word tokens generated as a result of preprocessing. Notice it doesn't contain any noise words.

bigramId	word1	word2
1	Draw	rectangle
2	rectangle	width
3	width	45
4	45	height
5	height	50
6	50	fill_color
7	fill_color	red
8	red	rotation
9	rotation	45
10	45	degree

Figure 30: Bigrams from the preprocessed output.

GlyphName	GlyphType	line.width	line.color	line.stroke	fill.color	tx	ty	th...	xs	ys
rectangle	rectangle	NULL	NULL	NULL	red	NULL	NULL	45	45	50

Figure 31: Final Glyph Profile that can be passed on to Manifold

This result contains all the properties and values as understood by Manifold, and can be directly passed on to it for glyph rendering purposes. Table 3 shows a list of SQL stored procedures that participate in the process of Text Recognition. It is noteworthy here that the process of text recognition explained above could also have been carried out using other programming languages like Java or Python which have very nice library for regular expressions for working with strings.

Table 3: SQL stored procedures for the process of text recognition

Stored Procedure Name	Function
RECOGNISER_GENERATE_GLYPH_PROFILE	Accepts word utterance and performs word preprocessing and bigram generation.
RECOGNISER_SPLIT_WORDS	Inserts distinct words in the table Word and returns the

	inserted word's id.
RECOGNISER_CHECK_PERMISSIBLE_VALUES	Verifies that a given property value is permissible for the given glyph type. For example the height and width should always be numeric.

4.3.1.3 Rendering Text Recognition Output on Manifold

In order to decouple the feature of text recognition from Manifold, its complete logic is separated from Manifold. The text parsing and manipulation was done using SQL Server, and was called through a SOAP Web service written in C#.NET. The front end was implemented as a menu item (custom glyph menu item in Insert menu). For more details on this interaction with Manifold, the reader is advised to see Chapter 6, section 6.6.3.

4.3.2 Speech Recognition

A speech interface can be used to directly issue commands to the domain. However, it could be used in a direct manipulation mode, such as commanding: *“Pick up the object X and start moving it north-east ... keep going ...keep going ... turn to the right ... stop.”*

Although there has been a great effort invested in trying to incorporate speech in human-computer interfaces, speech continues to play a minor role in HCI, and not because speech recognition is still imperfect. This may appear surprising, given that speech and language play the central role in human communication. Some challenges of speech-based interfaces are considered in [73, 74]. The greatest problem, in my opinion,

is that the computer is very unintelligent. All programs, despite their apparent complexity, have relatively simple knowledge and intelligence. Moreover, there is no knowledge sharing across different programs—all programs work independently—the only sharing is via the clipboard!

Because of this, humans still *operate* the computer, like a tool, rather than *communicating* to it like another intelligent being.

4.3.2.1 Speech Recognition Application in Manifold

In exploring the area of Speech recognition on Manifold, the main idea was to render glyphs on the Manifold Viewer by speaking in what has to be drawn. The creation of custom glyphs (through text recognition, section 4.3.1) by typing in what we need to generate is discussed in Chapter 6, section 6.6.3. Glyph generation through speech recognition was a direct derivative of this process.

We use Sphinx-4 [75] for the purpose of speech recognition. The integration of Sphinx in Manifold allowed transforming spoken words to text sentences which then could be passed on the Text Recognition application logic, as discussed in section 4.3.1, to perform proper manipulation and output a glyph profile which then could be rendered on Manifold. The main idea here is to pass whatever a user speaks to a backend application using a set of communication protocols as discussed in Chapter 3, section 3.2, since only the backend application could understand the meaning of semantic objects (Manifold doesn't know these and cannot understand them). Manifold just shows what it's told by the backend application, and the speech module directly speaks to a backend Sphinx application (not to Manifold).

This application of speech recognition is still in its naivety. However, after laying down the integration of Sphinx with Manifold, it could be extended in the future versions to build a more sophisticated application, where different user spoken actions could directly be mapped onto Manifold for direct manipulation of objects. For example, something similar to `ViewerMouseListener.java` (package `manifold.swing`) could be implemented to provide a listener to spoken words, independent of the Manifold viewer.

Chapter 5

Glyphs and Tools

The earlier versions of Manifold User Interface Framework anticipated certain glyphs and tools. These glyphs and tools though non-functional or non-implemented in the previous versions were essential in improving and enhancing the core functionality of the Manifold framework that could make it at par with the modern GUI tools available. To enumerate, these glyphs were *Text*, *Picture (Image)*, *Pin*; and tools were *Zoomer*, *Pinner*, *Grouper* and *Un-Grouper*.

5.1 Introduction: Tools, Manipulators and Controller

The *Tools* and associated *Manipulators* provided in the current application can be taken out and replaced with other Tools/Manipulators in different application contexts. Figure 32 summarizes the key characteristics of this tandem.

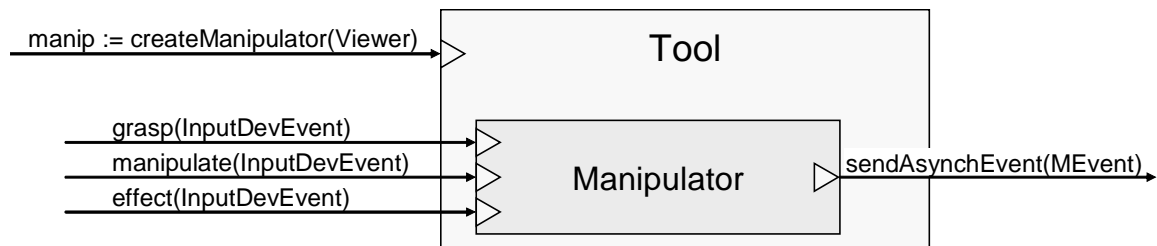


Figure 32: The main input and outputs of a Tool/Manipulator component

The developer must follow the `manifold.Tool` and `manifold.Manipulator` interfaces and implement their desired functionality for the grasp-manipulate-effect manipulation cycle.

The tools normally know very little or nothing about the glyphs they operate upon. For example, the `Creator` tool does not import any `Glyph` interface at all. `Deletor` imports `Glyph` to handle the list of glyphs scheduled for erasure. Both `Selector` and `Rotator` imports `Glyph2D` to access the glyph's transformation and `TransformGroup` to obtain the picking traversal service. `Selector` also uses the handle-movement simulation service. Obviously, this is a very basic knowledge and a broad range of spatial glyphs can be manipulated by the existing tools/manipulators. We believe that this demonstrates high degree of decoupling between the tools and objects (glyphs) on which they operate.

The implementation of the `Controller` interface (package `manifold`) is also related with tools/manipulators as discussed in Chapter 2, section 2.4. This is because the controller knows the *action verbs* of the event frames that the current application domain supports. The manipulator sets those verbs when creating the event frames for the domain. It is noteworthy here that the `Controller` is associated with the application domain only for the action verbs (add node, delete node, property query, and set properties), and not with the verbs that are used to describe the graphical attributes (see `EventFrame.java` in the package `manifold`). This minimal dependency of the `Controller` on the "application domain" was necessary for maintaining the core-functionality.

The tools are implemented in the package `manifold.impl2D.tools`. Table 4 below shows the Manifold tools and their functionalities.

Table 4: Manifold Tools and their Functionality

Tool Name	Functionality
Creator	Allows creation of new glyphs. (See <code>manifold.impl2D.tools.Creator.java</code>)
Deletor	Allows deletion of selected glyphs. (See <code>manifold.impl2D.tools.Deletor.java</code>)
Grouper	Allows grouping of multiple glyphs as a single poly-glyph (see <code>manifold.impl2D.tools.Grouper.java</code> , section 5.4.2)
Linker	Allows connecting of two glyphs by a linker line. (See <code>manifold.impl2D.tools.Linker.java</code>)
Pinner	Allows pinning a glyph to disable its translation at the pinned connector. (See <code>manifold.impl2D.tools.Pinner.java</code> , section 5.4.4)
Rotator	Allows rotation of a glyph by moving one of its handles in the desired direction. (See <code>manifold.impl2D.tools.Rotator.java</code>)
Selector	Allows selection of a single glyph by clicking, or multiple glyphs by drawing selection box around them. Also allows manipulating glyph transformation by manipulating their handles. (See <code>manifold.impl2D.tools.Selector.java</code>)
UnGrouper	Allows un-grouping of a poly-glyph to independent glyph objects. (See <code>manifold.impl2D.tools.UnGrouper.java</code> , section 5.4.3)
Zoomer	Allows zooming-in/out a glyph/canvas by pressing and moving a mouse cursor up and down. (see <code>manifold.impl2D.tools.Zoomer.java</code> , section 5.3.2)

5.2 Glyphs and Viewers

We believe that it is relatively easy to extend the “vocabulary” of glyphs that can be placed within the viewer canvas. Our glyphs pursue middle ground between what Bederson *et al.* [7] call *polylithic* and *monolithic* approaches to structured graphics. We introduce “shadow glyphs” that can be *composed* with visual glyphs to provide additional appearance or functionality. Although not entirely independent as nodes in polylithic

approaches, the shadow glyphs nonetheless provide separation of concerns and structured graphics aspects.

Glyphs know nothing about the tools/manipulators that operate on them. Generally, glyphs have minimal coupling with the rest of the Manifold framework. Glyphs are tightly coupled with the Viewer (package `manifold`) in which they will be displayed, and the viewer may provide “layout management” for them.

The glyphs are implemented in the package `manifold.impl2D.glyphs`.

Table 5 below shows the manifold glyphs and what they represent.

Table 5: Manifold Glyphs and their Representation on the Viewer

Glyph Name	Representation
Connector	Connector glyph that allows rendering glyphs like Pin, Slot, etc. (see <code>manifold.impl2D.glyphs.Connector.java</code>)
Connectors	Shadow glyph that manages the glyph's connectors. Connectors are optional. (see <code>manifold.impl2D.glyphs.Connectors.java</code>)
Grid	Grid figure, to be shown as a background of a glyph viewer. This glyph belongs to a category called <i>shadow glyphs</i> . Such glyph's only purpose is to shadow its <i>parent</i> glyph. As such, it should not be used for purposes other than the viewer background, and should not be possible to manipulate. (see <code>manifold.impl2D.glyphs.Grid.java</code>)
Highlighter	Highlights the glyph to let it distinguish itself graphically, for example, when it is selected. Draws the bounding box, with the handles for interaction, when a glyph is selected. The default color for both handles and the bounding box is <i>blue</i> . This glyph belongs to a category called <i>shadow glyphs</i> . Notice that the parent glyph does not keep shadow glyphs on its list of child glyphs. (see <code>manifold.impl2D.glyphs.Highlighter.java</code>)
Line	Line geometric figure (leaf glyph). As with other <i>Geometric Figure</i> 's, we assume that the <i>Line</i> 's shape is always centered on the origin (0, 0) of its local coordinate system. The endpoints of the <i>prototype</i> line segment are derived from <code>Point2D GeometricFigure</code> to have a <i>horizontal</i> line segment. (see <code>manifold.impl2D.glyphs.Line.java</code>)
Link	Link joins two connectors, which may be on the same glyph or on

	different glyphs. (see <code>manifold.impl2D.glyphs.Link.java</code>)
Picture	Picture/Image (leaf glyph) is used to render image graphics on the Manifold viewer space. Notice that this is a leaf glyph, but does not derive from <code>manifold.impl2D.GeometricFigure</code> . (see <code>manifold.impl2D.glyphs.Picture.java</code> , section 5.4.1)
Pin	Leaf glyph, to pin another glyph at one of its connectors in order to limit its translation/rotation. (see <code>manifold.impl2D.glyphs.Pin.java</code> , section 5.4.4)
Rectangular	Rectangular geometric figure (leaf glyph), derivative of <code>java.awt.geom.RectangularShape</code> , such as: <code>java.awt.geom.Ellipse2D</code> , <code>java.awt.geom.Rectangle2D</code> . We assume that the shape is always centered on the origin (0, 0) of its local coordinate system. If translated, that will be represented in the glyph's transformation. (see <code>manifold.impl2D.glyphs.Rectangular.java</code>)
Text	Text (leaf glyph) to render text on the Manifold viewer. Notice that this is a leaf glyph, but does not derive from <code>manifold.impl2D.GeometricFigure</code> . (see <code>manifold.impl2D.glyphs.Text.java</code> , section 5.3.1)

5.3 Previously Non-Functional Glyphs and Tools

Though included in the earlier versions of the Manifold, certain features like *Text Glyph* and *Zoomer* tool were non-functional. The text glyph just displayed the text “default text” and had an empty *Property Viewer* (discussed in Chapter 7) containing no *Property Editor*’s. The *Zoomer*, although being included in the UI, didn’t function. My first task was to get these glyphs and tools up and running.

5.3.1 Text Glyph

The package `manifold.impl2D.glyphs` is responsible for rendering the appropriate glyph on the UI. *Text* is a leaf glyph but does not derive from `manifold.impl2D.GeometricFigure` because unlike `GeometricFigures`

like `Rectangle2D` and `Line2D`, its shape cannot be transformed by selecting the connectors. The java class for drawing the text glyph on the work space is `Text.java` which inherits `manifold.impl2D.Glyph2D.java`. To render the user defined text on the Viewer, it was necessary to modify this class to support new property editors (*font editor* and *text editor*, which are discussed in detail in Chapter 7) and the event frame (to support new verbs). By including these new property editors and new verbs in the event frame, it became possible to render the text glyph as an editable glyph allowing changing properties like font size, face, and style. The new Event Frame verbs that were added to the class `EventFrame.java` (package `manifold`) were:

- `public static final String TEXT_FONT = "text.font";`

It was used to read the font state including the font size, face, and style, which were changed using the font editor (`manifold.swing.editors.FontEditor.java`, section 7.3.2).

- `public static final String TEXT_COLOR = "text.color";`

It was used to read the font color state that was changed using the color editor (`manifold.swing.editors.ColorEditor.java`).

- `public static final String TEXT_TEXT = "text.text";`

It was used to read the text state that was changed using the text editor (`manifold.swing.editors.TextEditor.java`, section 7.3.1).

The reason behind the text glyph not functioning properly was that a static text was defined in the XML file “tools.xml” (see the `manifold` package and source code for more details on this file) without any editors specified in “editors.xml” to modify this text. The code stub from “tools.xml” to render the default text is:

```
<void method="put">
    <string>text</string>
    <string>default text...</string>
</void>
```

As a result every time the user drew the text glyph this static text was rendered.

To display the appropriate text with different styles and text (which were changed in the corresponding text editors), the cached property of the glyph was read in the `Text.java` class. After adding the text editors with this glyph (see Chapter 7), the modified code stub from “editors.xml” file for Text glyph looked like this:

```
<!-- ***** Text Editor Panel ***** -->
    <void method="put">
        <string>text</string>
        <object class="manifold.swing.PropertyEditorsPanel">
            <void method="add">
                <object
                    class="manifold.swing.editors.FontEditor">
                        <void property="propertyName">
                            <string>text.font</string>
                        </void>
                    </object>
                </void>
            <void method="add">
                <object
                    class="manifold.swing.editors.TextEditor">
                        <void property="propertyName">
                            <string>text.text</string>
                        </void>
                    </object>
                </void>
            </object>
        </void>
    </void>
```

The rendering of glyph with new properties on the work-space was done by comparing the old states with the new states, only if the two states differ. The code stub (in the `draw()` method of `Text.java` class) that was added to implement this new feature is,

```
// Get the cached property from the editor and the one defined in XML.
String text_ = (String) cachedState.get(TEXT);
String textText_ = (String) cachedState.get(EventFrame.TEXT_TEXT);
```

```

//If new text has been set using the Text Editor, do update
if (textText_ != null && !textText_.isEmpty()) {
    text_ = textText_;
}

// Check if the glyph's font should be edited
Font textFont_ = (Font) cachedState.get(EventFrame.TEXT_FONT);
if (textFont_ != null) {
    graphics_.setFont(textFont_);
}

// Check if the glyph's color should be changed
Color textColor_ = (Color) cachedState.get(EventFrame.TEXT_COLOR);
if (textColor_ != null) {
    graphics_.setColor(textColor_);
}

```

Here the glyph's cached state is queried to read the new values (updated previously through the corresponding glyph property editors) belonging to the text glyph, and these new values are added to the glyph. The method `draw()` recursively traverses the scene graph to render and display the glyph in the viewer. The method takes `Traversal (package manifold)` type parameter having the current state of the traversal. `Traversal` implements the *Visitor* design pattern [8] for visiting a collection of glyphs. A traversal is passed to a glyph's traverse operation and maintains common information as well as the stack of information associated with each level of the traversal (see [4], section 4.3).

5.3.2 Zoomer

The *Zoomer* tool is responsible for zooming in/out the Manifold canvas. The user can zoom by clicking anywhere on the canvas and dragging down or up to zoom out and zoom in respectively. Zoomer provides only a visual feedback without changing glyphs actual dimensions (including the background Grid which acts as a shadow glyph). This is

different from glyph scaling where we change a glyph's transformation matrix to accommodate new scaling components.

The tool is implemented in the `manifold.impl2D.tools` package in the `Zoomer.java` class, and was included in the earlier versions of the Manifold, however was non-functional.

The `Zoomer.java` class extends to the class `BaseTool.java`, package `manifold.swing`. The class creates a `ZoomerManipulator` that converts the user interaction with the input device(s) to actions on the domain model (see Manipulator Chapter 2, section 2.4.1).

The main functioning of the Zoomer tool is in the method `manipulate()` which carries out the manipulation of the Manifold viewer. This method is typically called many times during a single interaction cycle. It also animates the manipulation in the viewer, so to provide direct visual feedback to the user, so the user feels confidence in what she/he is doing. It takes a parameter of type event object encapsulating the information received from the input device. This will typically be somewhat "processed" event, rather than a "raw" input device event.

The code stub that manipulates an event from the Zoomer tool is shown below:

```
//Get the JDesktopPane from the Grid glyph that acts as Manifold Viewer
viewerDesktopPane = ((Viewer2DImpl) currentGlyph.getViewer());
//Retrieve the JDesktopPane's (Manifold Viewer) graphics context
Graphics g = viewerDesktopPane.getGraphics();
Graphics2D g2 = (Graphics2D) g;

if ((currPointRel_.getX() < lastPointRel.getX() && currPointRel_.getY()
> lastPointRel.getY()) || (currPointRel_.getX() > lastPointRel.getX()
&& currPointRel_.getY() > lastPointRel.getY())) {
    //If mouse drag down then zoom out
    if (zoomOutD > 0) {
        zoomOutD -= 0.1;
    }
}
```

```

        g2.scale(zoomOutD, zoomOutD);

        viewerDesktopPane.paint(g2);
        viewerDesktopPane.validate();
    } else if ((currPointRel_.getX() < lastPointRel.getX() &&
currPointRel_.getY() < lastPointRel.getY()) || (currPointRel_.getX() >
lastPointRel.getX() && currPointRel_.getY() < lastPointRel.getY())) {
        //If mouse drag up then zoom in
        if (zoomInD < 2) {
            zoomInD += 0.1;
        }
        g2.scale(zoomInD, zoomInD);
        viewerDesktopPane.paint(g2);
        viewerDesktopPane.validate();
    } else {
        //Do Nothing
    }
}

```

Here, zoomOutD and zoomInD (variables of type double, both initially set to 1, as we keep the minimum and maximum scaling to 0 and 1 respectively to avoid negative or too much scaling) are decrease or increased respectively depending on whether the user drags mouse down or up relative to the current screen point (the point of mouse click). This is followed by scaling the entire Manifold Viewer's (a JDesktopPane [77]) graphic context. This provides a blown in or out effect to simulate the zoom effect as shown in Figure 33 and Figure 34.

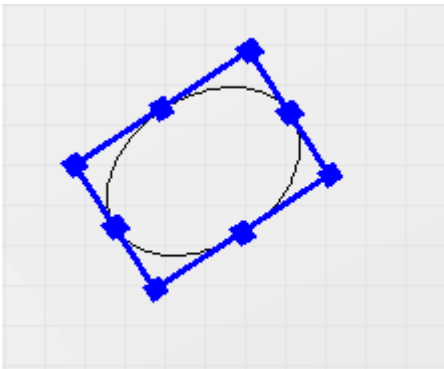


Figure 33: Original Glyph without applying the Zoomer Tool

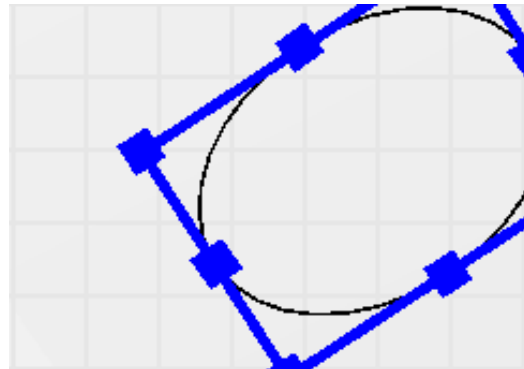


Figure 34: Zoomed In Glyph as a result of applying the Zoomer Tool to glyph in Figure 33

5.4 New Glyphs and Tools

After fixing the issues with the non-functional glyphs and tools, the idea was to implement more, new and better tools and glyphs, which could put Manifold at par with the modern user interfaces. Some of these ideas were taken from Microsoft Power-Point [9], however their implementation and concept has been different considering the fact that Manifold is implemented as a generic user interface. These new tools and glyphs have been implemented as separate classes independent of the previously implemented tools and glyphs.

5.4.1 Image Glyph

One of glyphs that were anticipated in the previous versions was the *Picture/Image Glyph*, where you can render any image/graphics on the Viewer space either retrieved from the database or the file system. This would allow rendering objects other than of type `Line2D` or `Rectangle2D` that earlier Manifold versions implemented.

The *Picture/Image* glyph, like other glyph types that Manifold support, resides in the `manifold.impl2D.glyphs` package. It is implemented as a separate class `Picture.java`. The Image is a leaf glyph; however it does not derive from the `manifold.impl2D.GeometricFigure` because unlike `GeometricFigures` like `Rectangle2D` and `Line2D`, its shape cannot be transformed by selecting the connectors.

The idea behind this glyph was to render an image in any size and apply transforms on it by just mouse click and dragging just like other glyphs (The idea was taken from Adobe Illustrator [14]).

The image was rendered, using the method `draw()`, as discussed previously, that takes parameter of type `Traversal`. Initially the image path is specified in the constructor `Picture()` of the class `Picture.java` that sets the default image by calling the function `getImagesFromDb()` (calls a SQL stored procedure that returns the name of the default images that are set in the database), explained next, which provides the names of the image(s) (the database table `Image` stores the name of the images that can be rendered on the Manifold viewer space. This name is combined with the file path that contains the physical location of the images on the system) to render as default on the viewer. The image(s) is set as a `BufferedImage`, which is a subclass of `java.awt.image` [15]. The `BufferedImage` [16] subclass describes an `Image` with an accessible buffer of image data.

The image is rendered on the viewer by using `drawImage()` method of `Graphics` [17] which is set in the `draw()` method of `Picture.java`. The image is changed using the *Image Editor* (see Chapter 7, section 7.3.3 for more details), by reading the cached state described by the new event frame verb `IMAGE_SOURCE`, set as (in `EventFrame.java`, package `manifold`):

```
//Event frame verb for image source
public static final String IMAGE_SOURCE = "img.source";
```

The code stub from method `draw()` for changing the new image read from the `ImageEditor` is:

```
// Check if the glyph's image file is changed.
File file_ = (File) cachedState.get(EventFrame.IMAGE_SOURCE);
if (file_ != null) {
    try {
        img = ImageIO.read(new File(file_.toString()));
    } catch (IOException e) {
    }
}
```

Here, the glyph's cached state is queried and the new image is read from the file if the cached state is not null.

5.4.2 Grouper

Grouper allows you to take different elements (graphics, charts, essentially anything but text) and attach them to each other. This means you can work with multiple objects as if they were one. You can move a grouped object as one thing. You can even resize a grouped object and all elements within the group are resized together and proportionally. In general, you perform any action on a grouped object that you can do on an individual element. It does not affect the physical placement of the objects relative to each other.

To understand the usefulness of the Grouper tool, think of a scenario where you would like to draw different objects on the graphical editor, and then give common attributes to all of them. For example if you would like to draw a graph, at a point you would not like to manipulate different nodes individually, but the whole-graph as a single smart-art, giving it a specific width, height, location on the user interface. If this is done individually on each of the items of the graph, it will require you to manipulate them 'n' (the number of independent components in the graph) number of times, which will be highly inefficient and in-elegant.

The idea of Grouper has been incorporated from Microsoft PowerPoint [9], and allows you to work faster. For example, you can manipulate multiple objects at a time, through the manipulation of a single object, and can change attributes like color, shape, size, translation, etc. You can also ungroup a group of objects at any time (using UnGrouper tool described in section 5.4.3), and regroup them later.

Manifold as a graphical editor didn't have this functionality in its earlier versions. However, with the addition of menus like *Insert Smart Art* (see Chapter 6, section 6.6.4 for more detail), the Grouper tool made a big sense. The scenarios in which this tool could be used extend to different applications that can run on Manifold. For instance, drawing smart-art (objects created by drawing glyphs separately and then grouping them to make a single art, like a smiley face). The Grouper tool would allow you to develop such graphics, that could be saved using the *Save menu item* (see Chapter 6, section 6.3.3), and retrieved later to fulfill these purposes. The need for such objects compelled us to develop the Grouper tool for Manifold user interface that would allow the user to forget the worries about performing similar operations on multiple graphics separately.

Design Issue 5.1: *The current implementation of Grouper doesn't allow grouping of Text/Picture glyph type. This is because these glyph types do not derive from manifold.impl2D.GeometricFigure and the current Selector implementation requires the bounding shape as Rectangle2D to be selected by drawing a selection box around glyphs. It should be extended in the future versions to select arbitrary shapes as well.*

5.4.2.1 Design: Grouper Tool

Grouper tool is implemented as a java class `Grouper.java` belonging to the package `manifold.impl2D.tools`.

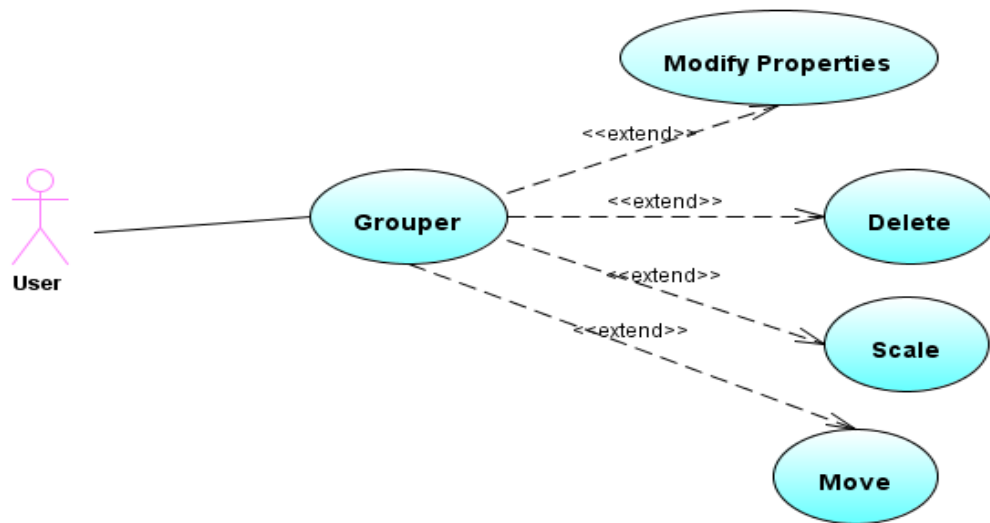


Figure 35: Use Case Diagram for Grouper

The development of this tool had certain requirements namely,

- User should be able to select multiple glyphs.
- It should only work if multiple glyphs are selected, as grouping a single glyph does not make any sense.
- The user should be able to get a visual feedback if the glyphs are grouped.
- And finally, the user should be able to perform manipulations like moving, scaling, deleting, changing properties/attributes of the grouped glyphs as a single entity.

The use-case diagram in Figure 35 shows these requirements in more detail (the current implementation doesn't support rotate functionality). In order to confer with these requirements, the design of the Grouper tool has to be a little complex, and yet flexible, so that minimal or no change should occur in other Manifold classes.

The motivation behind developing this tool comes from the Composite design pattern [8], which allows building a complex object out of elemental objects and itself like a tree structure such that all the objects could be treated as a single instance of an object. The resultant object in context with Manifold is a poly-glyph, which means a glyph that can have child glyphs. The poly-glyph is implemented in class `TransformGroup.java` belonging to package `manifold.impl2D`.

The Grouper tool extends to the Selector tool (`manifold.impl2D.tools.Selector.java`) and creates a `GrouperManipulator`. The `grasp(event_)` method creates a Grouper glyph of type rectangle when a selection is made around the glyphs to be grouped with the same coordinates as that of the selection box (the transform matrix is set according to the mouse coordinates). This happens only if the current selection mode is `PICK_RECTANGLE` (see `Selector.java`). This grouper glyph would allow the manipulation of all the underlined selected glyphs as one object. In other words all the selected glyphs would be a part of this grouper glyph. The `manipulate(event_)` method adds this grouper glyph node to the domain.

The actual grouping happens in the `effect(event_)` method which gathers all the selected glyphs, and sets the current selections to all the underlying glyphs. If there is only one underlying glyph, it forces the grouper glyph to be deleted (as grouping only a

single glyph would be same as manipulating it independently). If number of selections is greater than one (two including the grouper glyph), all the selected glyphs are added as children to the grouper glyph using the method `addChild(Glyph)`, that adds the argument glyph as a child to the calling glyph (see `Glyph.java`). The code stub that does this is:

```
//Make a poly-glyph (see Composite Design Pattern), if number of
//selections (including the grouper) is greater than 2
if (glyphs_.length > 2) {
    //The grouper glyph is the last glyph of the selections (n-1)th
    final Glyph grouperGlyph = glyphs_[glyphs_.length - 1];

    for (int i_ = 0; i_ < glyphs_.length - 1; i_++) {
        grouperGlyph.addChild(glyphs_[i_]);
    }

    Glyph[] childGlyphs_ = new Glyph[0];
    childGlyphs_ = grouperGlyph.getChildren();
    //Set the minimum bounding shape for the grouper glyph
    setMinimumBoundingShape(grouperGlyph, childGlyphs_);
}
```

It then calls the method `setMinimumBoundingShape(grouperGlyph, childGlyphs_)`, taking arguments as the grouper glyph itself, and all its child glyphs. It then sets the minimum bounding shape of the grouper glyph so that it just envelopes all the child glyphs.

Design Issue 5.2: *The implementation of Grouper as a tool instead of making it a menu item (like in Microsoft PowerPoint [9]) was done since the current version of Manifold doesn't allow multiple selections by pressing the control key (as keyboard actions are still not supported). The only way to select multiple glyphs is to draw a rectangular bounding box around the glyphs that requires the grasp, manipulate, effect cycle supported only by tools. The current design choice is also helpful in a way as it*

allows the user to group closely located glyphs quickly. To allow implementation of Grouper as a menu item to allow selections of arbitrary glyphs, keyboard actions should be implemented in the future versions. It should be noted here that in the current implementation the selection box should completely cover the underlying glyph to add it to the current selections.

The sequence diagram in Figure 36 shows the grasp, manipulate, and effect cycle of the Grouper tool.

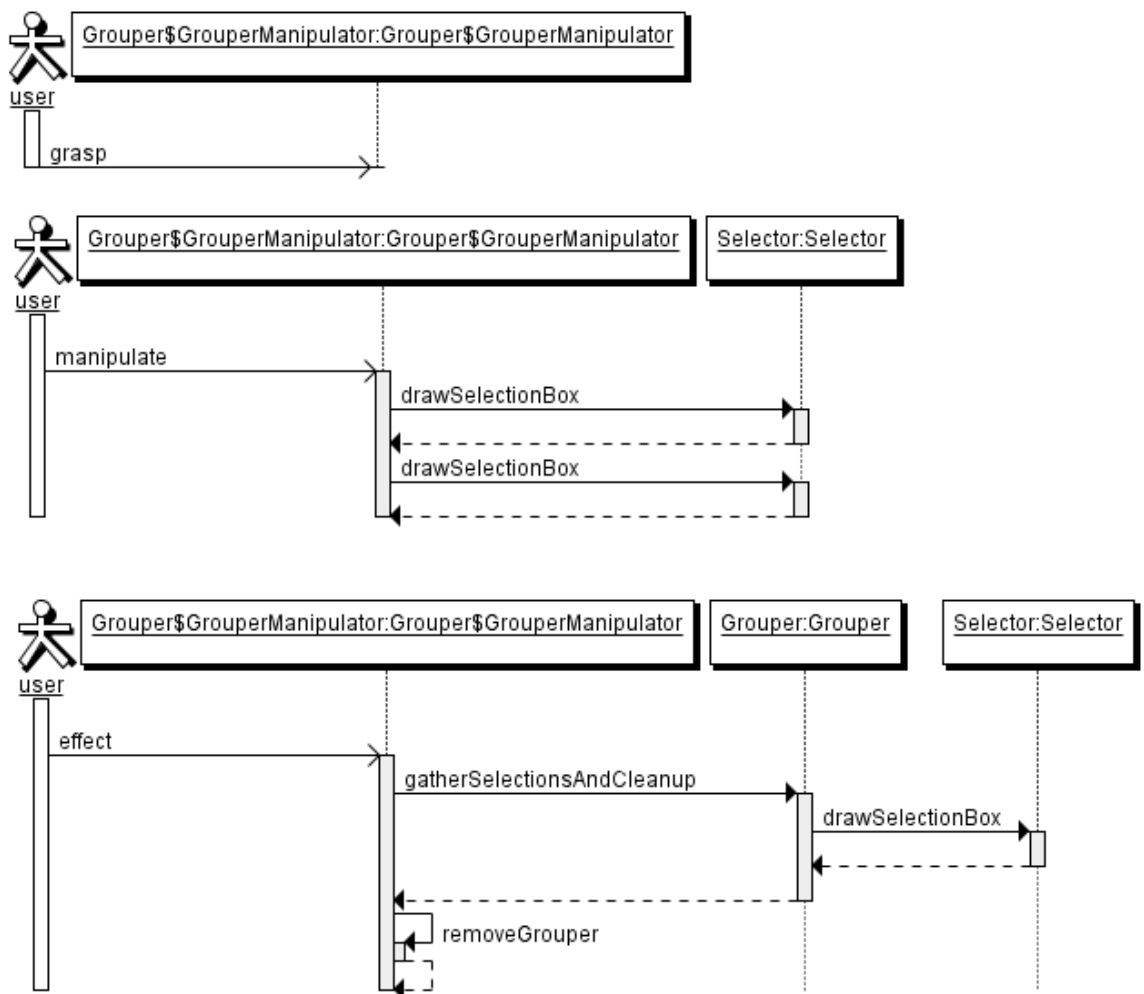


Figure 36: UML Sequence Diagram showing the grasp, manipulate, effect cycle of Grouper Tool

Figure 37, Figure 38, and Figure 39 explains the entire process of grouping.

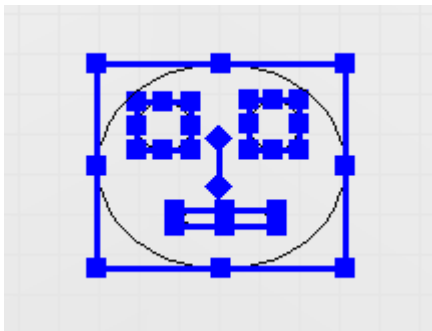


Figure 37: Five glyphs drawn independently on the Manifold Viewer.

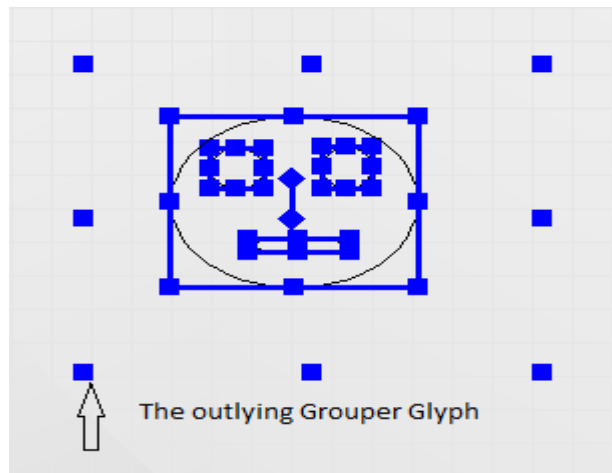


Figure 38: The glyphs from Figure 37 after grouping.

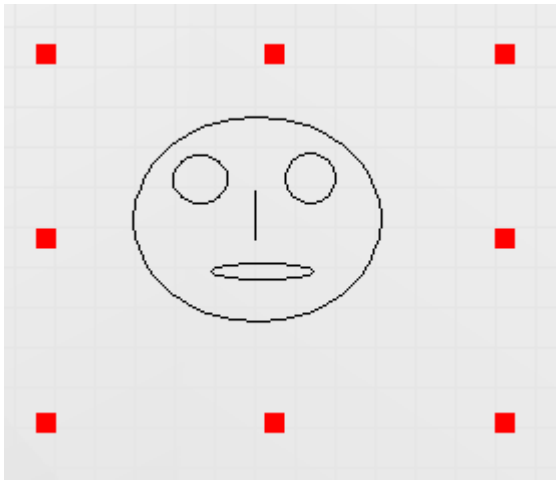


Figure 39: The independent glyphs can now be treated as a single smart art (a smiley here). Notice the connectors become visible when you scroll over the object.

5.4.2.2 Interpreting Grouped Glyphs

Once the glyphs have been grouped, the job is only half done, as there is a need to tell other Manifold classes that manipulate on glyphs, to behave accordingly with the grouped glyphs as well. One of these classes is the *Selector* tool, in which the *Selector*

Manipulator has to be instructed to manipulate on the grouper glyph so that the corresponding effects are transferred on to the child glyphs as well. Similar changes have to be made in the *Deletor* tool as well, where if a grouper glyph is selected to be deleted, all the child glyphs should also be deleted.

The classes and the respective changes that have to be incorporated to understand the Grouper glyph are shown in Table 6.

Table 6: Changes made in Manifold classes to support Grouper tool.

Class	Modification
Selector	The <code>manipulate(Object)</code> method belonging to the <code>SelectorManipulator</code> inner-class has been modified to transfer the manipulations like “scaling” and “translation” to the child glyphs when it occurs on the grouper glyph.
GeometricFigure	The <code>draw(Traversal)</code> method has been modified to make the grouper glyph invisible by forcing its scaling and translation matrix to be a null matrix. This is done by reading the <code>EventFrame GROUPER_GLYPH</code> . If it is not null, it implies that the current glyph is a Grouper glyph, and it should be made invisible. Another change in the same class has been made on how the Grouper glyph is saved. The reader is advised to see Chapter 6, section 6.3.3 for details on how this is accomplished.
Deletor	To incorporate deleting grouped glyphs, the current scene graph is made parent of the child glyphs i.e. the current parent of the grouper glyph, followed by deleting the individual glyphs one by one.

5.4.3 Un-Grouper

The *Un-Grouper* tool performs the opposite actions of the Grouper tool. It separates individual children glyphs from the poly-glyph. The Un-Grouper tool is implemented in the java class `UnGrouper.java` (package `manifold.impl2D.tools`) and extends to `Selector.java`. It creates an

`UnGrouperManipulator` (a java inner-class) that extends to the `SelectorManipulator` (`Selector.java`) and redefines only what happens in the `manipulate(event_)` and `effect(event_)` manipulation cycles. The `manipulate(event_)` enforces the selection of only type `PICK_POINT` (a single click). When the user clicks on the Grouper glyph, it gathers all the children of the composite glyph and removes them from the parent Grouper glyph. It then sets the parent of all the children glyph to the current scene graph (the parent of the main Grouper glyph). Finally it sends a delete action verb to the `ControllerImpl` (package `manifold`) to delete the Grouper glyph that invisibly encircled the children glyphs before. It should be noted here that if the underlying glyph that needs to be ungrouped, is not a Grouper (poly-glyph) then it won't perform any action on the glyph.

5.4.4 Pinner

The *Pinner* tool is used to “pin” a glyph at one of its connectors to restrain its translation with respect to the pinned connectors only. In other words Pinner is used to force zero degrees of freedom to a glyph at one of its connectors (a shadow glyph that maintain glyph's connectors). The pinned glyph can still be rotated or scaled at the location where it is pinned with the center point of transformation being the glyph's bounding box. The Pinner tool is implemented in the java class `Pinner.java` (package `manifold.impl2D.tools`) and extends to `BaseTool.java` (package `manifold.swing`). It creates a `PinnerManipulator` (a java subclass) that extends to the `Manipulator` (package `manifold`) and defines only what happens in

the `grasp(event_)` and `effect(event_)` manipulation cycles. The `grasp(event_)` gets the current connector under the mouse position and draws a Pin glyph at the particular connector. The Pin glyph specifies the semantics to reduce the degree of freedom of the underlying glyph at the pinned connector (see `manifold.impl2D.glyphs.Pin.java`).

Figure 40, Figure 41 and Figure 42 shows the working of the Pinner tool.

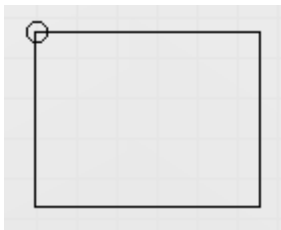


Figure 40: A pinned glyph of type rectangle

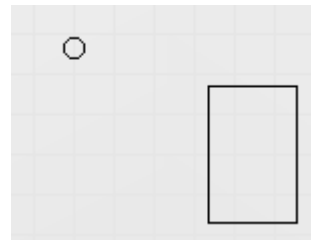


Figure 41: Glyph's translation being disabled at the pinned connector

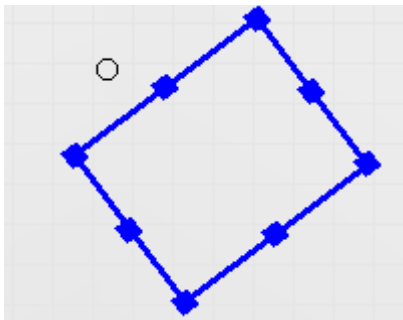


Figure 42: The pinned glyph could still be rotated at the pinned connector

Chapter 6

Menu Bar

A menu bar is a horizontal strip where a list of available computer menus is housed for a certain program. A menu provides a space-saving way to let the user choose one of several options.

Nearly all programs have a menu bar as part of their user interface. It includes menu items and options specific to the particular program. Most menu bars have the standard File, Edit, and View menus listed first. The *File* menu includes options such as Save and Open File..., the *Edit* menu has items such as Undo, Copy, Paste, and Select All, while in the *View* menu you'll find viewing options such as changing the layout of open windows (view full screen, minimize window). Programs, such as Microsoft Word [9], also include menu options such as *Insert*, *Format*, and *Font* which you will most likely not find in a Web browser's menu bar. But a Web browser may contain menu options such as *History* and *Bookmarks*, which you will not find in a word processing program.

Many items located within the menu bar often have keyboard shortcuts that enable you to choose menu options by just pressing a key combination. The menu items available with menu bar can each fire a Command or open a cascaded menu. The menu bar is a fundamental part of the graphical user interface (GUI), so it is worth your time to get familiar with it. You may even discover features you did not know about before.

6.1 Introduction

The Manifold menu bar is very similar to the menu bar of other available GUI's. It has options (menu items) to perform the above discussed operations in context of glyphs. The menus available in the Manifold menu bar are:

- File: Contains the following menu items-
 - New Workspace – allows opening a new (empty) workspace.
 - Open Document – allows opening a previously saved Manifold document.
 - Save Selection(s) – allows saving the glyph selection(s) into a database.
 - Save Document – allows saving the entire document into a database.
- Edit: Contains the following menu items-
 - Undo – allows undoing the last performed action.
 - Redo – allows redoing the last performed action.
 - Select All – allows selecting all glyphs currently present in the Manifold viewer.
 - Select None – allows removing the current selections.
- View: Contains the following menu items-
 - Full Screen – allows viewing the Manifold viewer in full screen.
 - Minimize – allows minimizing the Manifold workspace (similar to the minimize button at top right of the Manifold workspace).
 - Map Viewer – allows opening Manifold with a map viewer.
- Insert: Contains the following menu items-

- Geometric Figure – a cascaded menu that contains the following menu items-
 - Rectangle – allows inserting previously saved glyphs of type rectangle.
 - Ellipse – allows inserting previously saved glyphs of type ellipse.
 - Line – allows inserting previously saved glyphs of type line.
- Image – allows inserting new images from the file system.
- Custom Glyph – allows inserting a custom glyph by typing into a command box what has to be drawn.
- Smart Art – allows inserting a previously saved smart art (a collection of glyphs that were grouped by using the Grouper tool, section 5.4.2).

6.2 Design: Menu Bar

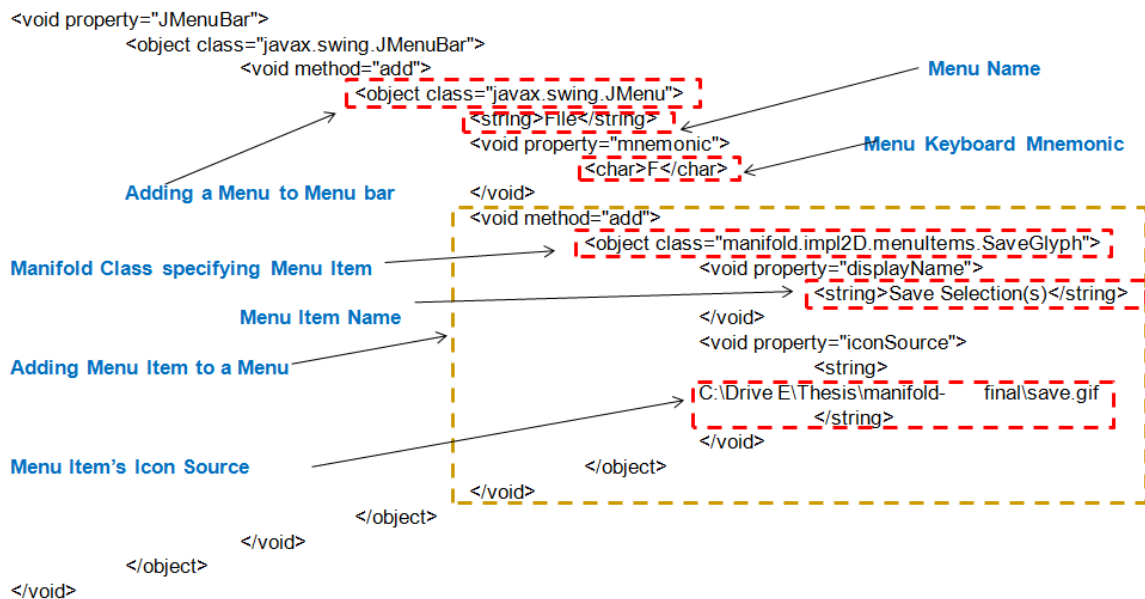


Figure 43: XML definition of Manifold Menu Bar

The menu bar is defined in the file “menuItems.xml” provided with the packaged source code. This XML file provides the list of menu and the corresponding menu items present in the Manifold menu bar. A code stub from this file that draws the file menu is shown in Figure 43.

The implementation of menu items required addition of some new classes that were added to the Manifold source package. These classes are described in Table 7.

Table 7: New Manifold classes for implementing a functional Menu Bar

Class Name	Functionality
<code>manifold.MenuItem.java</code>	Describes the interface for menu items.
<code>manifold.MenuItemsModel.java</code>	It manages a collection of menu items used for performing glyph related operations like Insert/Save on the Manifold Viewer or actions like View Full screen etc.
<code>manifold.swing.BaseMenuItem.java</code>	Java Swing specific base class for specifying JMenuItem [21] through the menu-bar. For e.g. inserting clip-art objects. It contains the method <code>setViewerForRendering()</code> that sets the viewer for rendering purposes. It retrieves the hierarchy of the menu-item to get its root pane, from where it gets the content pane of Manifold, which is the main workspace for drawing purposes. This is done as we want to render the graphics on the Manifold viewer (see <code>Viewer2DImpl.java</code>).
<code>manifold.SavedItem</code>	An interface that defines the items that are saved into the database. This class can be implemented, for example to retrieve saved glyphs from the persistence. The functionality could be increased in future to retrieve saved tools/property editors, etc.
<code>manifold.swing.TableSavedGlyphs.java</code>	Concrete class that is used to retrieve saved glyphs from the database.
<code>manifold.swing.TableSavedDocuments.java</code>	Concrete class that is used to retrieve saved document and all the glyphs belonging to it from the database.
<code>manifold.swing.TableModel.java</code>	Java Swing specific class that provides an Abstract Table Model for a JTable [24], such that it

	allows selecting a particular row. When clicked on, it gathers all the columns with their values.
<code>manifold.swing.TableSelectionListener.java</code>	Implements a selection listener for a <code>JTable</code> . When a user clicks on a particular row on the <code>JTable</code> , it retrieves the selected property-value pairs from various columns. It then calls the <code>renderSelectedItemOnViewer()</code> of class <code>TableSavedGlyphs.java</code> method to render the selected object on to the Manifold viewer space.

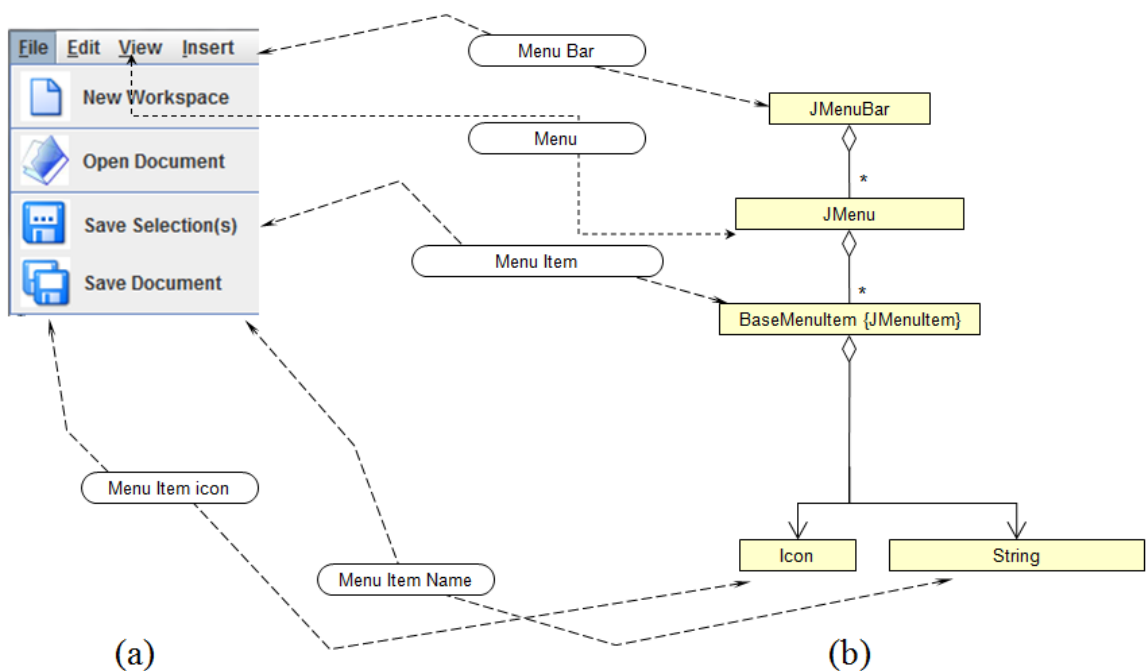


Figure 44: Composition of Menu Bar: (a) The screen rendering; (b) The UML class diagram

Figure 44 shows the current menu bar implementation. All the menu items are implemented in separate classes belonging to the package `manifold.impl2D.menuItems`. The classes extend to `BaseMenuItem` (package `manifold.swing`), and implements interface `ActionListener`. It is added to the `ActionListener` via the `addActionListener()` method in the constructor of

the class. When an action event occurs, the object's `actionPerformed()` method is invoked that performs menu item specific operations that would be discussed in detail in the next few sections for each menu item.

6.3 File Menu

A file menu is a common drop-down menu that includes commands for file operations, such as Open, Save, and Print. The Manifold File Menu contains four menu items, as discussed in section 6.1, which are New Workspace, Open Document, Save Selection(s), and Save Document.

6.3.1 New Workspace Menu Item

The *New Workspace* menu item is used to open a new instance of the Manifold workspace as a part of the single Manifold process. New workspace menu item is implemented in class `manifold.impl2D.menuItems.NewWorkspace.java`. The `actionPerformed()` method calls the method `openNewInstance()` having the following code stub:

```
private void openNewInstance() {
    try {
        XMLDecoder dec_ = new XMLDecoder(
            new BufferedInputStream(
                new FileInputStream(xmlFileDraw2D)));
        // Read all the root-level elements in the XML document.
        try {
            while (true) {
                Object result_ = dec_.readObject();
            }
        } catch (ArrayIndexOutOfBoundsException aiobex_) {
            // Ignore -- no more objects to read!
            // (This is the only way to detect the last element.)
        }
        dec_.close();
    }
}
```

```

    } catch (Exception ex_) {
        Debug.exception("manifold.util.Application", "main", ex_,
            null);
    }
}

```

Here, the “draw2D.xml” file path is specified to the XMLDecoder [51] object, which instantiates the Manifold classes as specified in the XML file and opens a new workspace (without affecting the current workspace, which also remains open).

Design Issue 6.1: *The new workspace runs as a part of the single application process thread (Main). Hence when the user closes a workspace, the entire process is terminated, and all the open workspaces are closed. In future versions a more effective way to open a new workspace, like tabs (as the one used in modern web browsers), should be thought about.*

6.3.2 Open Document Menu Item

The *Open Document* menu item allows opening a previously saved workspace document (discussed next in sections 6.3.3, 6.3.4) so that all the glyphs belonging to that document could be directly rendered on Manifold viewer. The `actionPerformed()` method has the following code stub:

```

//calls the base class method to set the viewer for rendering of
//document glyphs
    setViewerForRendering();

//Populate jTable with all the saved documents
TableSavedDocuments tableSavedDocuments = new
TableSavedDocuments(ITEM_TYPE, viewer);
tableSavedDocuments.getSavedItemFromDatabase();

tableSavedDocuments.addTableToPopUp(this.getParent().getParent(), TOP,
TOP);

```


Here, as soon as the user clicks the Open Document menu item, the viewer is set by calling the base class (`BaseMenuItem.java`) method `setViewerForRendering()`. This is followed by setting the current item type to “document”. The “document” item type and the current viewer are then passed to the class `manifold.swing.TableSavedDocuments.java` whose method `getSavedItemsFromDatabase()` gets all the previously saved documents from the database by calling a SQL stored procedure `GET_DOCUMENT`. This procedure accepts one parameter, the `documentId` and outputs a `SQL ResultSet` which is then displayed in a `JTable` [24] using the method `addTableToPopUp()`. The `JTable` [24] is drawn, with a custom table model [25] and with a custom `ListSelectionListener` [25]. This is used so that the user can click on any row of the table to select a particular glyph, from the retrieved glyphs. Table Model is implemented in the class `TableModel.java` (package `manifold.swing`), and the `ListSelectionListener` is implemented in the class `TableSelectionListener.java` (package `manifold.swing`). Figure 45 shows the output table when the user clicks on the Open Document menu item.

documentId	documentName
1	MyFirstWorkspace
2	MySecondWorkspace
3	My3rdWS
4	My3rdWS1
5	My3rdWS2

Close

Figure 45: Output when the user clicks on the Open Document Menu item

When the user selects one of the rows from the table, the document (and all the glyphs that were saved as a part of it) corresponding to the selection is rendered on the Manifold viewer. This is done by calling the method `getAllGlyphsInDocument(documentId)` by the class `TableSelectionListener` that gets all the glyphs belonging to the passed document id and renders them on the Manifold viewer. The rendering process would become clear in section 6.6 when we discuss the Insert menu.

6.3.3 Save Selection(s) Menu Item

The most complex menu item available in the Manifold menu bar is the Save Selection(s) that allows saving any glyph(s) as a clip-art directly into a remote server database with separate application logic so that they can be retrieved and re-rendered on the Viewer using the Insert menu. This menu item can save all possible glyph kinds viz. leaf glyphs (rectangle, ellipse, etc.) and poly-glyphs (glyphs having children) selected on the Manifold viewer. However, when the user wants to retrieve them back, they appear as their specific types (geometric figures viz. rectangle, ellipse, etc. or smart-art for poly-glyphs) as explained in the Insert Menu (section 6.6).

SaveSelection(s) menu item is implemented in the class `manifold.impl2D.menuItems.SaveGlyph.java`. The

`actionPerformed()` method has the following code stub:

```
//invoke the base class method to set the viewer for rendering of
//clip-Art
setViewerForRendering();

//get the current selection(s) i.e the selected glyphs on the viewer
this.currentNodeId = viewer.getSelectionsModel().getSelections();
```

```

String typedGlyphName = (String) JOptionPane.showInputDialog(
    this,
    "Type in the glyph name",
    "Glyph Name",
    JOptionPane.PLAIN_MESSAGE,
    null,
    null,
    "");

if (!typedGlyphName.isEmpty() && typedGlyphName != null) {
    Hashtable slots_ = null;
    for (int i = 0; i < currentNodeId.length; i++) {
        slots_ = new Hashtable();
        slots_.put(EventFrame.VERB,
            ControllerImpl.SET_PROPERTIES);
        slots_.put(EventFrame.NODE_ID, currentNodeId[i]);
        slots_.put(EventFrame.SAVE_GLYPH, typedGlyphName);
        viewer.getController().sendAsyncEvent(new
            EventFrame(slots_));
    }
}

```

Here, as soon as the user presses the Save Selection(s) button (a JMenuItem), the viewer is set by invoking the base class (BaseMenuItem.java) method `setViewerForRendering()`. The node id's of the selected glyphs from the viewer are then stored in a string array `currentNodeId` and a `JOptionPane` [18] dialog is made visible, to allow the user to enter the glyph name. When the user enters the glyph name and presses the "OK" button on the dialog, the glyph(s) name, if not empty, is added to the `EventFrame`. This allows the `draw()` method (see `manifold.impl2D.GeometricFigure.java`) of the selected glyph(s) to query the event frame and get the glyph name to save the selected glyphs by calling the method `saveGlyph()`, if the glyph name is not null. The code stub to do this is:

```

// Check if the glyph should be saved.
String saveGlyphState_
    = (String) cachedState.get(EventFrame.SAVE_GLYPH);
if (saveGlyphState_ != null) {
    saveGlyph(saveGlyphState_);
}

```

In the `saveGlyph()` method, the glyph's `cachedState` and `transform` are passed to a SQL stored procedure. The name of the stored procedure is selected based on whether the current glyph is a grouper glyph (i.e. it contains child glyphs, a poly-glyph) or an individual leaf glyph. The code stub to do this is:

```
Hashtable currentNodeProperties = this.getCachedState();
String cacheProps_ = currentNodeProperties.toString();
String transform_ = this.getTransform().toString();

String storedProcedureName = "";
//If the saved glyph is a grouper glyph, save the child glyphs as a
//single object
if (isGrouper(currentNodeProperties)) {
    if (this.getChildren().length > 0) {
        Glyph[] childrenGLyphs = null;
        childrenGLyphs = this.getChildren(); //Get all the child of
        //this poly-glyph

        for (int i_ = 0; i_ < childrenGLyphs.length; i_++) {
            Glyph2D testGlyph = null;
            testGlyph = (Glyph2D) childrenGLyphs[i_];

            cacheProps_ +=
                " |" + testGlyph.getCachedState().toString();

            transform_ +=
                " |" + testGlyph.getTransform().toString();
        }

        storedProcedureName =
            "{call INSERT_GROUPEM_CLIP_ART(?,?,?,?,?,?)}"
    } else { // if it is a leaf-glyph
        storedProcedureName = "{call INSERT_CLIP_ART(?,?,?,?,?,?)}"
    }
}
```

Here, the stored procedures `INSERT_GROUPEM_CLIP_ART` or `INSERT_CLIP_ART` are called depending on the condition whether the current glyph is a grouper glyph or not (the `isGrouper()` method retrieves the event frame `GROUPEM_GLYPH` from the glyph's cached state to know whether it is a poly-glyph or not). The stored procedures takes in four input parameters as `cachedState`,

transform, `glyphName` and `documentId` (used to save glyph as a part of a document, section 6.3.4), and outputs `success` and the inserted glyph's id, if the save glyph was a success. The `INSERT_GROUPEL_CLIP_ART` calls the `INSERT_CLIP_ART` procedure for each of the child glyphs present in the poly-glyph.

The main steps of this stored procedure can be enumerated as:

- Parse the input cached states and transform.
- Retrieve meaningful glyph properties and the corresponding values from these text utterances provided by the cached state.
- Store the glyph name, along with its type in the table `Glyph`. Also store document id if the current glyph is saved as a part of a document (using the Save Document menu item, section 6.3.2), `NULL` otherwise.
- Store the glyph profile i.e. the property values in the table `GlyphProfile`.
- If save is successful return “true”, else “false”.

These steps could be understood better with the database diagram shown in Figure

46. The database diagram shows the following tables:

- `GlyphType`: Stores a list of Manifold glyph types viz. Rectangle, Ellipse, Line, Image, Text, etc. It could support new glyph types in future just by inserting a new row with the new glyph type name.
- `Document`: Stores a document name (discussed in section 6.3.4).
- `Glyph`: Stores the name, type and parent (of the child glyphs belonging to a Grouper glyph) of the saved glyph. It also relates to a document if it was saved as a part of it, otherwise the `documentId` field is `NULL`.

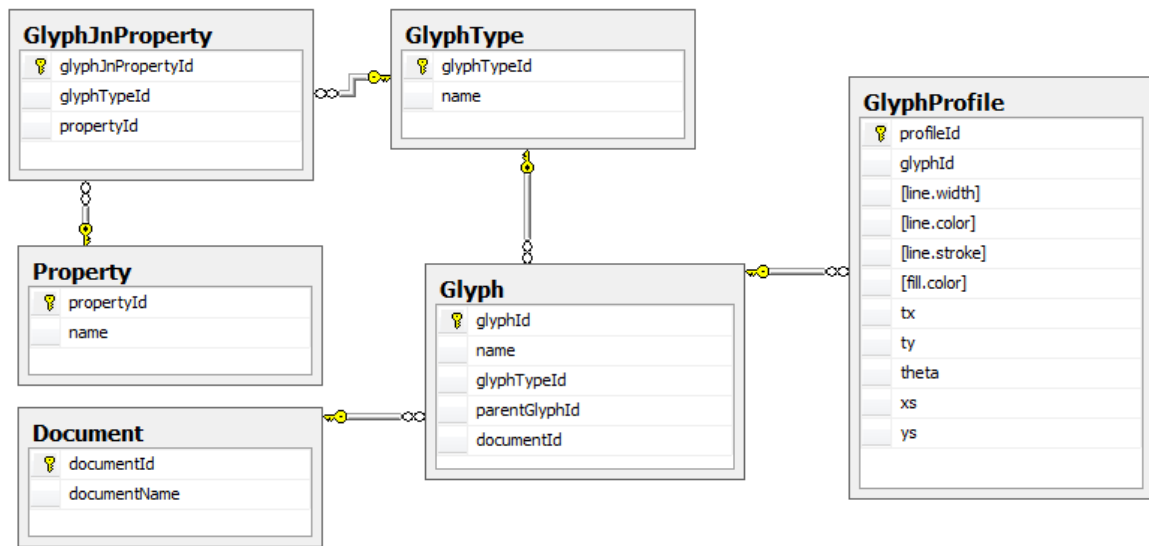


Figure 46: Database diagram showing the table relationship to save glyphs (leaf- or poly glyphs) or document.

- **Property**: Stores the name of properties that can be supported by Manifold glyphs.
- **GlyphJnProperty**: Specifies a many-to-many relationship between a glyph type and property (as different glyphs can have same properties).
- **GlyphProfile**: Stores the glyph profile i.e. the saved glyph along with its entire property-value pair. It also stores the parent of the glyphs if they belong to a Grouper glyph. This makes it possible to render the glyph again on the viewer when called by the Insert menu (see section 6.6).

Figure 47 shows the actions after the user clicks the Save Selection(s) menu item. When the user enters the glyph name and presses “OK” on the JOptionPane [18], the corresponding entries are saved in the database tables, through the process discussed above. Figure 48 and Figure 49 shows the database table with new entries.

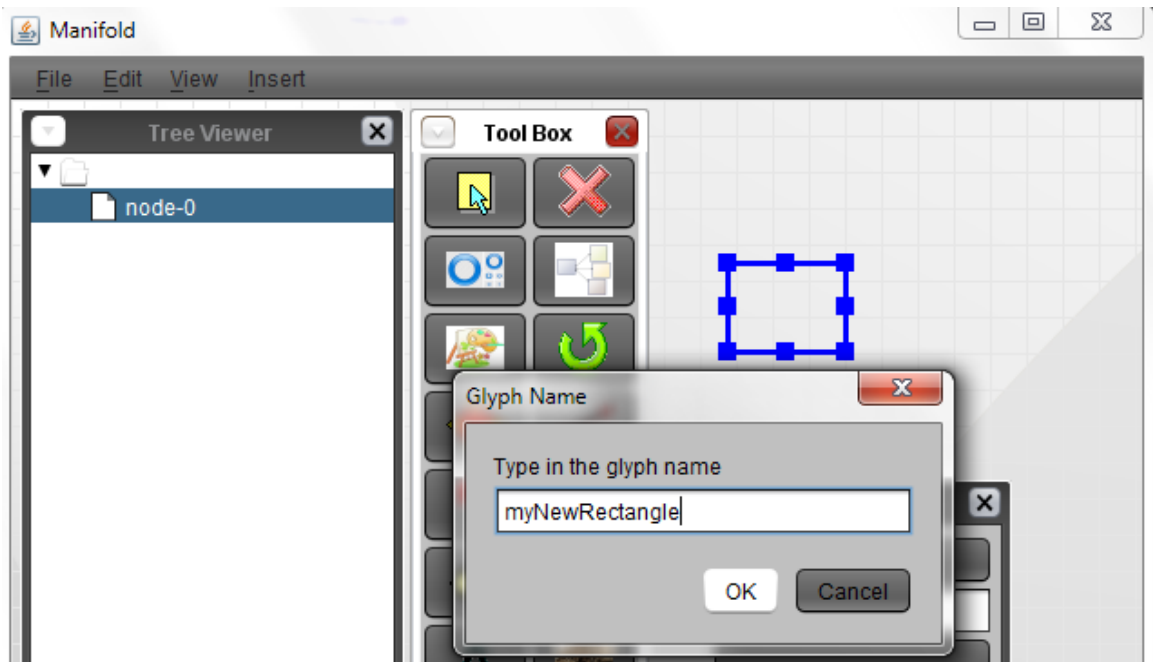


Figure 47: A JOptionPane [18] appears when the user presses the Save Selection(s) menu item.

	glyphId	name	glyphTypeId	parentGlyphId
▶	40	myNewRectangle	2	NULL

Figure 48: New entry created in table Glyph. Notice the parentGlyphId is NULL as this is not a poly-glyph.

	profileId	glyphId	line.width	line.color	line.stroke	fill.color	tx	ty	theta	xs	ys
▶	1	40	5.0	java.aw...	java.awt....	java.a...	516.5	44.0	0.0	79.0	74.0

Figure 49: New entry created in table GlyphProfile. Notice the glyph properties and transform (tx, ty, theta, xs, ys) stored as text attributes in the table.

To appreciate how the glyphs are stored as text fields inside the database, the reader should see the stored procedures `INSERT_CLIP_ART` and `INSERT_GROUPER_CLIP_ART` provided with the source package that implements the process discussed above.

Design Issue 6.2: *At an early implementation stage, the save feature was implemented as a Property Editor, instead as a Menu Item. However, it was being argued that (confering to the standards of all the modern GUI tools available) since it doesn't modify any of a glyph's properties, it should be implemented as a menu item. It should be noted here that an EventFrame value "SAVE_GLYPH" is set to store the glyph's name (so that it can be read for saving, as discussed next). This value is cleared from the cached state as soon as the glyph saving process is completed.*

Design Issue 6.3: *The Save Selection(s) menu item stores a glyph's properties and transform in database tables as fields, instead of saving it as a geometric object (as done in conventional GUI's) in the database or on the file system. This was done as there was an increase in performance when dealing with saving strings rather than objects. Also this would allow saving space, as text requires less space than an object when saving.*

6.3.4 Save Document Menu Item

The Save Document menu item is used to save the current workspace document so that all the glyphs in it could be retrieved at a later stage without the need to redraw all the glyphs that we want to use from a previously built Manifold document.

Save Document menu item is implemented in the class `manifold.impl2D.menuItems.SaveDocument.java`. The `actionPerformed()` method calls a function `saveDocument(String)` that

accepts the document name (by which it could be retrieved using the Open Document menu item, as shown in Figure 45) as the argument and saves it in the database table Document (refer to the database diagram in Figure 46). It calls the stored procedure SAVE_DOCUMENT that takes in one input parameter as documentName, and outputs documentId (the inserted document's id in database table Document). If a document is saved using the same name again, then the stored procedure deletes the previously saved glyphs (and their profiles) of the current document, before saving the new glyphs.

To tell the application domain that the currently present glyphs on the viewer form a part of a document, the SAVE_DOCUMENT EventFrame is set, which can then be read by the draw() method of a glyph to know whether a glyph should be saved as a part of the document. The draw() method recursively traverses the scene graph through which a glyph's cached state could be queried at any point.

Design Issue 6.4: *The Save Document saves all the glyph types except Text and Image as the current implementation doesn't support saving of these glyph types. The current application logic to save glyphs could be easily extended to non-geometric figures like text and image in the future versions to support their saving/retrieval as well.*

6.4 Edit Menu

An edit menu is a common drop-down menu that includes commands for changing (editing) the contents of documents, such as Cut, Copy, and Paste. The Manifold Edit Menu contains four menu items as discussed in section 4.1 above which

are Undo (non-functional in current version), Redo (non-functional in current version), Select All and Select None.

6.4.1 Select All Menu Item

The Select All menu item allows the selection of all the glyphs present on the Manifold viewer. It is implemented in the class `manifold.impl2D.menuItems.SelectAll.java`. The

`actionPerformed()` method has the following code stub:

```
public void actionPerformed(ActionEvent e) {
    //invoke the base class method to set the viewer.
    setViewerForRendering();

    //set the current selection(s) i.e the glyphs on the viewer
    Glyph[] allGLyphs = viewer.getSceneGraph().getChildren();
    String[] selections = new String[allGLyphs.length-1];

    //start from 1, ignoring the shadow glyph Grid that is the
    //background
    for(int i=1;i<allGLyphs.length;i++){
        selections[i-1] = allGLyphs[i].getId();
    }
    viewer.getSelectionsModel().setSelections(this, selections);
}
```

Here, as soon as the user clicks Select All (a `JMenuItem` [21]), the viewer is set by invoking the base class (`BaseMenuItem.java`) method `setViewerForRendering()`. This is followed by querying the scene graph of the viewer to provide the list of all glyphs present on the viewer space. This allows setting the `SelectionsModel` (package `manifold`) selections by passing the node id of all the glyphs present on the viewer. It should be noted that the background `Grid` (package

`manifold.impl2D.glyphs`) is a shadow glyph and acts as a background of a glyph viewer. Hence, it is not added to the list of the selections.

6.4.2 Select None Menu Item

The Select None menu item allows de-selecting all the currently selected glyphs present on the Manifold viewer. It is implemented in the class `manifold.impl2D.menuItems.SelectNone.java`. The `actionPerformed()` method gets the glyphs present on the viewer and sets their `selected` property to `false` (see method `manifold.Glyph.java#setSelected(boolean)`).

6.5 View Menu

A view menu is a common drop-down menu that includes commands for viewing options such as changing the layout of the open windows like full screen, minimize, restore, web layout, etc. The Manifold View Menu contains three menu items as discussed in section 6.1 which are Full Screen, Minimize, and Map Viewer.

6.5.1 Full Screen Menu Item

The Full Screen menu item allows maximizing the Manifold window. It is implemented in the class `manifold.impl2D.menuItems.FullScreen.java`. The `actionPerformed()` method calls the method `maximizeWindow()` that

allows setting the Manifold's `JFrame` [23] property `setExtendedState()` to `MAXIMIZED_BOTH` (state used to maximize the frame horizontal and vertical) .

6.5.2 Minimize Menu Item

The working of minimize menu item (belonging to the class `manifold.impl2D.menuItems.Minimize.java`) is similar to that of Full Screen menu item discussed above in section 6.5.1, with the only difference being that the `JFrame`'s [23] property `setExtendedState()` is set to `ICONIFIED` (state used to iconify i.e. minimize the frame).

6.5.3 Map Viewer Menu Item

Since the early days of navigation, maps have played a vital part of commerce, military, and other day to day requirements. From visualizing geographically to searching, maps have always been an integral part of the human civilization. Many websites now provide all sorts of interesting data that can be searched, indexed, and visualized geographically.

To expand the features of Manifold, so that it could be used in map applications, a new kind of visual map feature was required to be added. *Map Viewer* is a special type of viewer that can be used to draw graphics with respect to locations on a map on the Manifold viewer space. It acts as a placeholder for a map and the Manifold glyph viewer, supporting input device events for both of them interchangeably. This means that a user could manipulate glyphs (just like as on a normal Manifold viewer) and simultaneously

visualize them on a map (which works independently and supports its own input device events). It can also be attached with a back-end database that specifies information about a particular location which could be visualized on Manifold. The importance of such a kind of editor comes in the cases, when the user wants to save a particular glyph to specify certain location parameters. For example, the user could create a bar/pie chart using Manifold, and then group it as a single object using the Grouper tool discussed in Chapter 5, section 5.4.2, and subsequently save this object. This object can then be attached to a particular location on the map, and when the user wants to search about that location at a later time, the object could be directly rendered. In case the information has changed during this time the user can modify the object and save it again.

The Map Viewer could also prove to be very efficient in cross functional planning and execution. It can be used for interactive visualizations and collaborative decision support if a number of user share the same Manifold workspace hosted on a central server, like the commanders and the officers on the battlefield. This feature can allow you to get into other people's head and let them get into yours in real-time collaboration environment. The users could easily zoom into various levels of the map using information at various levels, giving them a better understanding of situations and possible outcomes.

There are a large number of software and applications available today for the process of real-time collaboration and planning on a map. However, doing so in context of Manifold gave an important advantage of how highly demanding and efficient applications can easily be built on Manifold and yet be highly portable and light-weighted in terms of application resources. The regular Manifold glyphs can be overlaid

and manipulated on the map viewer just like the normal Manifold viewer, providing interactive visualizations like construction of graphs. This can be done using the following main features of glyph manipulation on Manifold:

- Draw nodes and edges (using the Manifold *Creator* tool).
- Editors to color nodes and edges (using the Manifold *Property Editors*).
- Dragging and dropping of nodes and edges (using the Manifold *Selector* tool).
- Labeling of nodes under user control (using the Manifold *Text* glyph).
- Interactive pan/zoom and rotate functionality (using the Manifold *Zoomer/Rotator* tools).

Figure 50 illustrates one such possible interaction of the Map Viewer built on the top of Manifold and utilizing its pre-existing features to draw graph components that connects various cities in the US. The nodes could be colored differently and the edges can be provided with different stroke types to represent various kinds of possible links between the cities.

This is only one such visual representation of what could be accomplished with the Map Viewer, and depending on the requirements and needs it could be extended to support multiple scenarios like visualizing 3-D components, etc. The application developers only need to specify new 3-D glyph types and 3-D map tiles to fit the tandem. At this time I will leave it to the application developers on how they want to utilize and develop their applications in front of Manifold framework, the only key feature being the generic nature of Manifold.

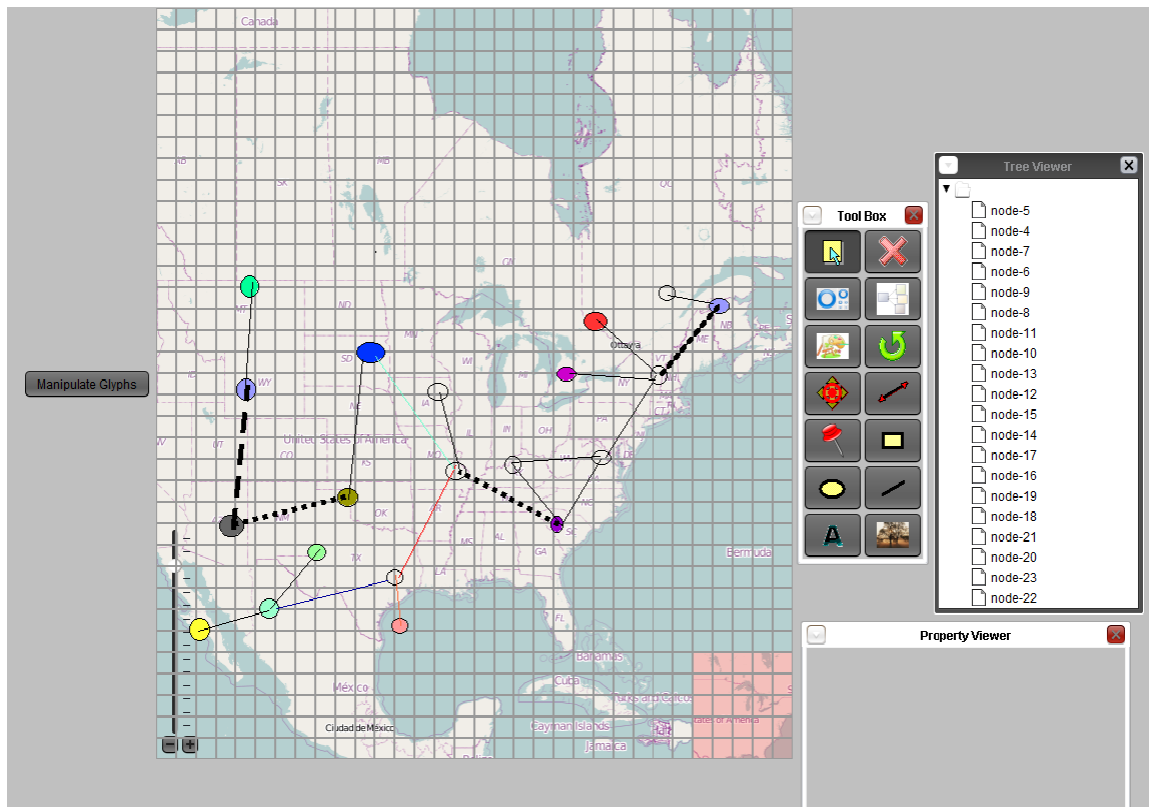


Figure 50: The Manifold Map Viewer displaying a graph connecting various locations on the map using Manifold glyphs.

6.5.3.1 Design: Map Viewer

The *Map Viewer* is implemented as a menu item in the class `manifold.impl2D.menuItems.MapViewer.java`. The

`actionPerformed()` method opens a new workspace (as done in the *New Workspace* menu item, section 6.3.1) by instantiating Manifold classes defined in the file “draw2DMap.xml”.

The “draw2DMap.xml” file redefines the way the content pane of the application should be laid out. There are 3 main classes that were added to the package `manifold.swing` in order to provide the functionalities of the Map Viewer as

discussed in the previous section. These classes and their functionalities are defined in Table 8.

Table 8: Manifold classes used to layout the Map Viewer

Class Name	Functionality
ViewerMapImpl	This class defines the new content pane (extends to JPanel [28]) of the application instance that supports the Map Viewer. It acts as a placeholder for the map (a JXMapKit [52]) and the Manifold glyph viewer (MapOverlayGlyphViewer). It supports the mouse events for both the map and the glyph viewer (manifold.swing.ViewerMouseListener) that can both register themselves in order to receive event notifications. This allows direct manipulation so the user can manipulate the glyphs using different tools and receive real-time visual feedback about how his/her activity affects the glyph's appearance with respect to the map. It uses a JToggleButton [54] to switch between the two types of mouse events.
MapOverlayGlyphViewer	This acts as a viewer for the Manifold glyphs and extends to manifold.swing.Viewer2DImpl (same as the main Manifold glyph viewer).
MapLayoutManager	This is the basic layout manager, doesn't do much, but size all the components (JXMapKit and MapOverlayGlyphViewer) to fill whole parent layer (currently a JLayeredPane [53])

Figure 51 shows the various components of the Manifold Map Viewer. The main component is the JXMapView [52] which is an open source (LGPL) Swing component created by the developers at SwingLabs [55]. At its core, the JXMapView is a special JPanel that knows how to load map tiles from an image server. It supports mouse events that are used to pan and zoom the map viewer's surface. The JXMapView's API provides details of how to convert coordinates to pixels, cache tiles, and stitch them together on screen.

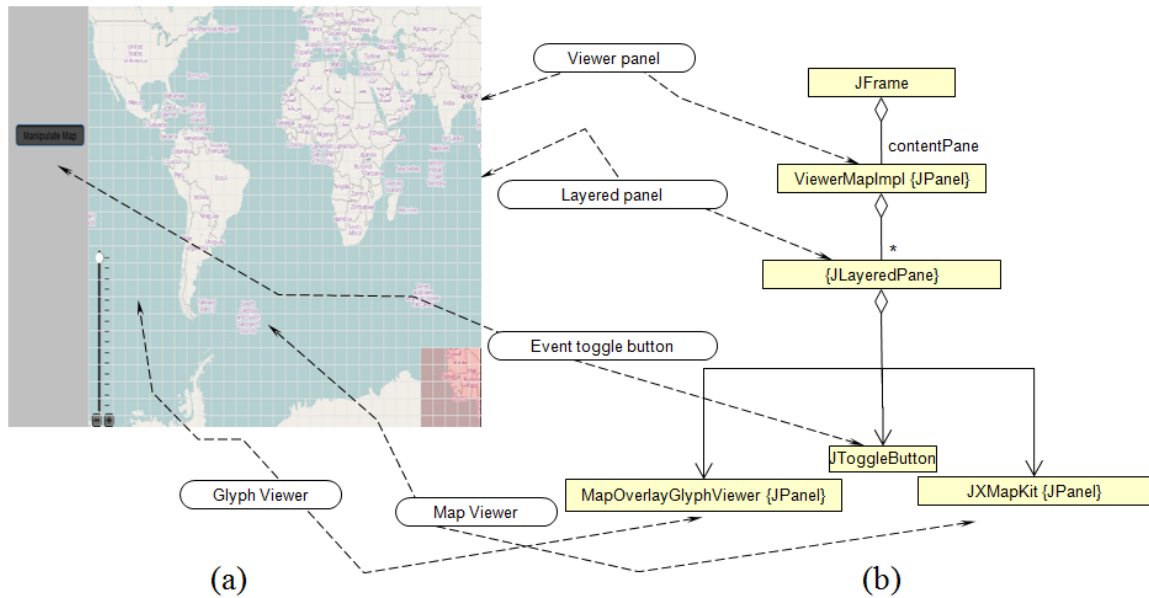


Figure 51: Composition of Map Viewer: (a) The screen rendering; (b) The UML class diagram

The main challenge in implementing the map viewer was to capture the events both with respect to Manifold glyphs as well as the one supported by the `JXMapKit`. To overcome this issue, there was a need to introduce a new placeholder that could accommodate both Manifold viewer and the `JXMapKit`'s map viewer. This was done by adding a new class `ViewerMapImpl.java`, belonging to package `manifold.swing`, which provides an interactive map, and allows direct manipulations on it. This map is added to a `JLayeredPane` [53] and handles the map's mouse events. On top of the `JLayeredPane` another viewer is added that supports *direct manipulation* of graphical elements (glyphs) rendered within it. Direct manipulation means the user can manipulate the glyphs using different tools and receive real-time visual feedback about how his/her activity affects the glyph's appearance with respect to the map. This can be a very important feature in the situations when you would like to

draw glyphs at specific locations on the map, with both the glyph and map listening to different mouse events at the same time. A listener interested in mouse events for glyph manipulation, `manifold.swing.ViewerMouseListener`, registers itself in order to receive event notifications. As a result the map viewer was made capable to support different event types, which could be controlled using a `JToggleButton` [54] as shown in Figure 51. Figure 52 shows the sequence diagram representing this tandem.

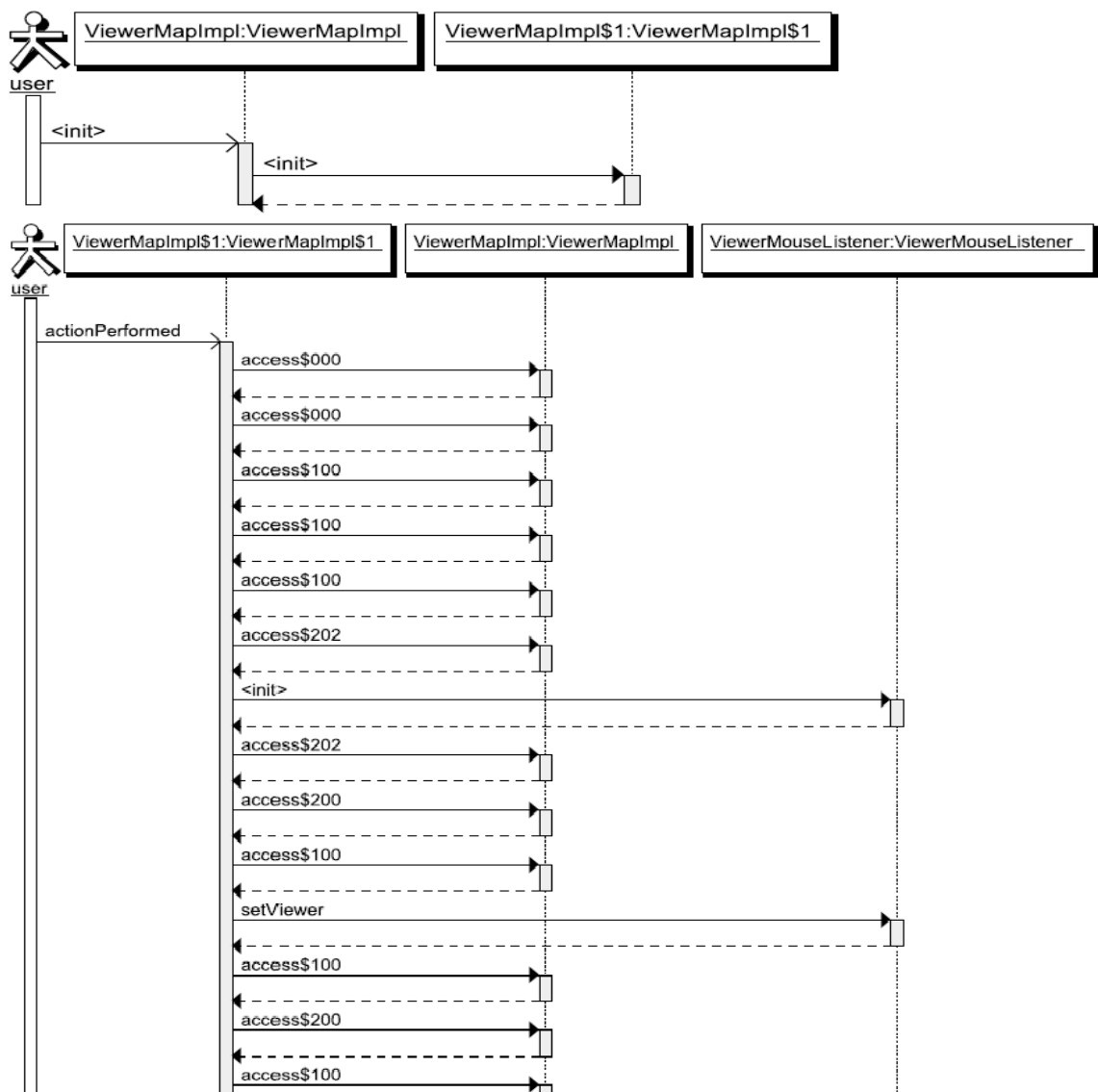


Figure 52: Sequence Diagram showing `ViewerMapImpl<init>` and `ViewerMapImpl.actionPerformed` cycles

6.6 Insert Menu

An insert menu is a common drop-down menu that includes commands for inserting objects, such as Picture, Smart Art, and Clip Art. Insert is an important menu in any graphical editor, as it allows user to draw pre-defined (and pre-saved) images/graphics directly, without the need to draw them from the scratch, which saves a lot of time and improves the performance of the GUI as well. For instance when the users works on glyphs on Manifold, at some point they want to save the current state of the glyph (leaf or poly-glyph) and come back later to work on it again. Saving glyphs has already been discussed in section 6.3.3.

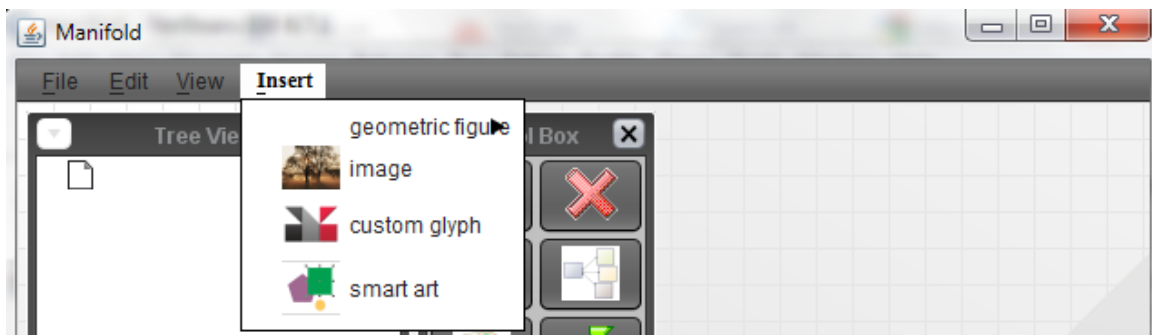


Figure 53: The Manifold Insert Menu

The Manifold Insert menu contains three menu items as discussed in section 4.1 which are Image, Custom Glyph and Smart Art. It also contains a cascaded menu, Geometric Figure that contains three menu items which are Rectangle, Ellipse and Line. Figure 53 shows the Manifold Insert Menu. It is noteworthy to mention here that the Insert Menu lists individual glyphs that were saved before using the Save Selection(s) menu item (section 6.3.3), and not the glyphs that were saved as a result of saving the entire document (section 6.3.4).

6.6.1 Geometric Figure Menu

The geometric figure menu is a cascaded menu inside the Insert Menu. It allows inserting the previously saved geometric figures (leaf glyphs) on the Manifold viewer space. The three types of geometric figures are Rectangle, Ellipse and Line (which are three glyph types in Manifold). Inserting a previously saved leaf-glyph could save a lot of time in comparison to creating it from scratch and providing it properties like fill color, line color, stroke, size, etc.

6.6.1.1 Rectangle Menu Item

The Rectangle menu item allows inserting previously saved glyphs of type Rectangle on the Manifold viewer space. It is implemented in the class `manifold.impl2D.menuItems.ClipArtRectangle.java`. The

`actionPerformed()` method has the following code stub:

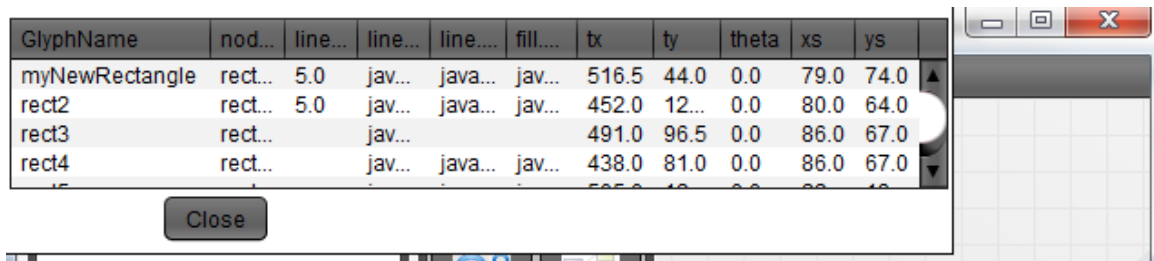
```
//calls the base class method to set the viewer for rendering of
//clip-Art
setViewerForRendering();

//Populate JTable with all the saved glyphs of type rectangle.
TableSavedGlyphs tableSavedGlyphs = new TableSavedGlyphs(CLIP_ART_TYPE,
viewer);

tableSavedGlyphs.getSavedItemFromDatabase();
tableSavedGlyphs.addTableToPopUp(this.getParent().getParent(), TOP,
TOP);
```

Here, as soon as the user clicks the Rectangle menu item, the viewer is set by calling the base class (`BaseMenuItem.java`) method `setViewerForRendering()`. This is followed by setting the current clip art type to “rectangle”. The “rectangle” clip art type and the current viewer are then passed to the

class `manifold.swing.TableSavedGlyphs.java` whose method `getSavedItemsFromDatabase()` gets all the previously saved glyphs of type “rectangle” from the database by calling a SQL stored procedure `GET_CLIP_ART`. This procedure accepts two parameters, the `clipArtType` and the `parentGlyphId` (used for inserting smart art, i.e. the grouped objects, discussed in section 6.6.4) and outputs a SQL `ResultSet` which is then displayed in a `JTable` [24] using the method `addTableToPopUp()` (as discussed in section 6.3.2). Figure 54 shows the output table when the user clicks on the Rectangle menu item.



GlyphName	nod...	line...	line...	line...	fill...	tx	ty	theta	xs	ys	
myNewRectangle	rect...	5.0	jav...	java...	jav...	516.5	44.0	0.0	79.0	74.0	
rect2	rect...	5.0	jav...	java...	jav...	452.0	12...	0.0	80.0	64.0	
rect3	rect...		jav...			491.0	96.5	0.0	86.0	67.0	
rect4	rect...		jav...	java...	jav...	438.0	81.0	0.0	86.0	67.0	

Figure 54: List of saved glyphs of type “rectangle” retrieved from the database, along with its profile (properties and transform).

When the user selects one of the rows from the table, the glyph corresponding to the selection is rendered on the Manifold viewer. This is done by calling the method `renderSavedItemOnViewer()` by the `TableSelectionListener` that creates a new node of type “rectangle” and passes the property-value pair (retrieved from the glyph profile in the table) to the `Controller` and consequently the new glyph is rendered on the viewer.

6.6.1.2 Ellipse Menu Item

The working of Ellipse menu item is similar to that of the Rectangle menu item as discussed in section 6.6.1.1, with the only difference being that the `CLIP_ART_TYPE` is set to “ellipse”. This menu item is implemented in class `ClipArtEllipse.java`.

6.6.1.3 Line Menu Item

The working of Line menu item is similar to that of the Rectangle and Ellipse menu item as discussed in section 6.6.1.1 and 6.6.1.2 respectively, with the only difference being that the `CLIP_ART_TYPE` is set to “line”. This menu item is implemented in class `ClipArtLine.java`.

6.6.2 Image Menu Item

The Image menu item allows inserting images present on the file system on the Manifold viewer. It is implemented in the class `ClipArtImage.java`. The `actionPerformed()` method has the following code stub:

```
//set the viewer for rendering of clip-Art
setViewerForRendering();

JFileChooser chooser = new JFileChooser();
// Note: source for ExampleFileFilter can be found in FileChooserDemo,
// under the demo/jfc directory in the Java 2 SDK, Standard Edition.
int returnVal = chooser.showDialog(this, "Select an Image");

if (returnVal == JFileChooser.APPROVE_OPTION) {
    File file_ = chooser.getSelectedFile();
    createImageGlyph(file_);
}
```

Here, as soon as the user clicks the Image menu item, the viewer is set by calling the base class (`BaseMenuItem.java`) method `setViewerForRendering()`. This is followed by calling the method `createImageGlyph(File)` that sends `EventFrame` actions to the domain model to generate glyph of type image and renders the image being selected from the `JFileChooser` [40].

6.6.3 Custom Glyph Menu Item

The Custom Glyph menu item is used to insert a user specified glyph on the Manifold viewer space. Here by custom glyph I mean that the user could type in what he wants to draw, and the corresponding text will be converted into a glyph and rendered on the Manifold Viewer. For example the user could write “Draw a rectangle with width 50 and height 45 with a rotation of 45 degree”. The custom glyph type converts this input text utterance into meaningful commands that Manifold could understand, and consequently the specified glyph is drawn on the Viewer. This, however, is true only for leaf-glyph types (rectangle, ellipse, line) in the current version and could be extended in future easily to include poly-glyphs as well. This process forms a part of *Text Recognition* that was discussed in detail in Chapter 4, section 4.3.1. The user is advised to refer to the same for an in-depth analysis of this process.

The Custom Glyph is implemented as a menu item just like Rectangle/Ellipse menu items. The only difference is that it doesn’t display the already saved glyphs with their profiles in a table for selection purposes, rather it asks for a user input to render the

glyph. It is implemented in the class `ClipArCustomGlyph.java`. The `actionPerformed()` method has the following code stub:

```
//call the base class method to set the viewer for rendering of
//clip-Art
setViewerForRendering();

//Take input from the user in a JOptionPane
String typedGlyphText = (String) JOptionPane.showInputDialog(
    this,
    "Type in what would you like to generate",
    "Glyph Text",
    JOptionPane.PLAIN_MESSAGE,
    null,
    null,
    "");

if (typedGlyphText != null) {
    this.fromWebService = true;
    textRecogniserWebService(typedGlyphText);
}
```

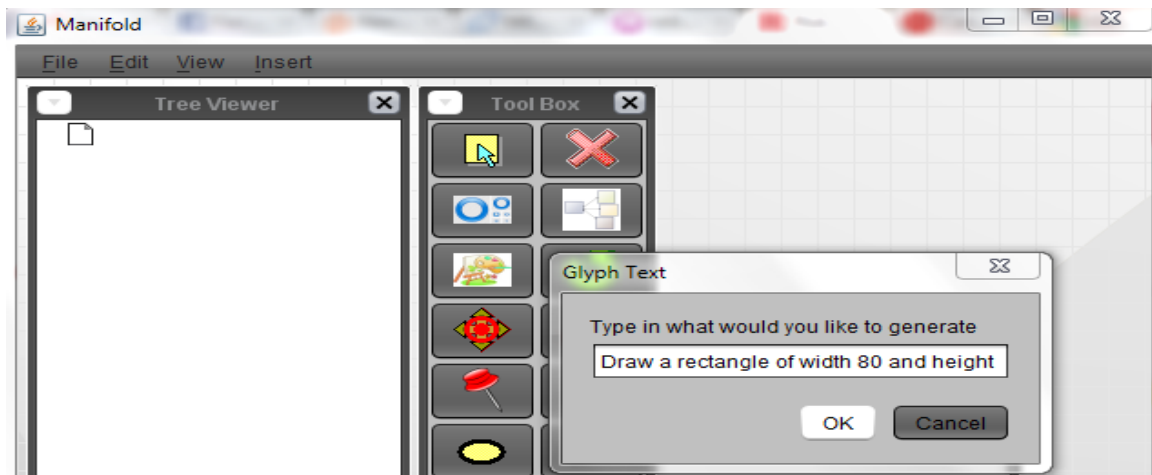


Figure 55: Manifold Custom Glyph. The user enters input in a JOptionPane.

Here, as soon as the user clicks the custom glyph menu item, a `JOptionPane` [18] is shown to take the input from the user, and if the typed text is not empty, it calls the method `textRecogniserWebService(String)`, which takes a string input and passes it to the SOAP [13] web service. The call to the web service is implemented

through an inner class `BasicWebServiceClient`. See Chapter 3, section 3.2 for a thorough discussion on protocols used for information exchange between Manifold and backend applications.

Figure 55 shows this process where the user enters text, “Draw a rectangle of width 80 and height 100 with rotation 45” to generate a rectangle.

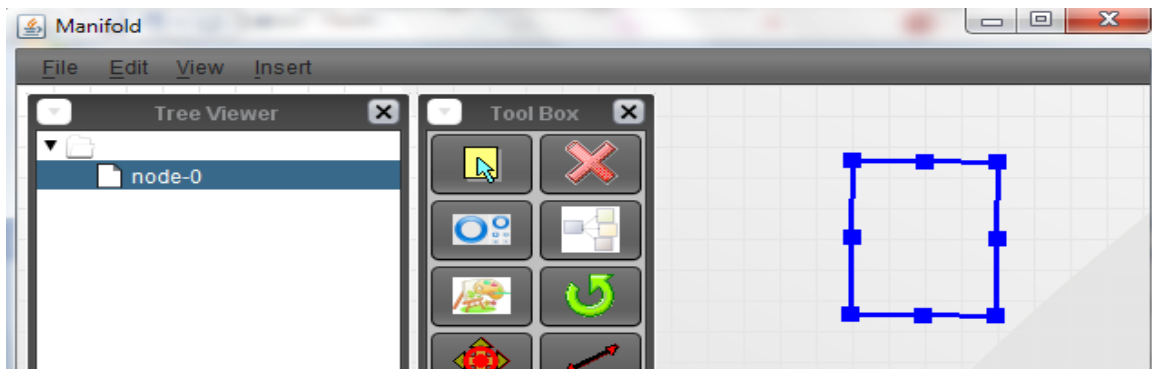


Figure 56: A new glyph gets created and drawn on the Manifold viewer as a result of Text Recognition

When the user clicks “OK” on the input dialog, the SOAP web service outputs the following property value pair to Manifold:

“GlyphName=rectangle;nodeType=rectangle;line.width=NULL;line.color=NULL;line.stroke=NULL;fill.color=NULL;tx=NULL;ty=NULL;theta=45;xs=80;ys=100”

As a result the corresponding glyph (if text is properly recognized, see details in Chapter 4, section 4.3.1) is created on the Manifold viewer. Figure 56 shows this.

The web service is responsible for taking in input from Manifold, passing it to a remote application server, and returning back Manifold the result (the Glyph profile). The generation of Glyph profile through the process of Text Recognition was discussed in detail in Chapter 4, section 4.3.1. Figure 57 shows the interaction of Manifold with the web service and the remote application server.

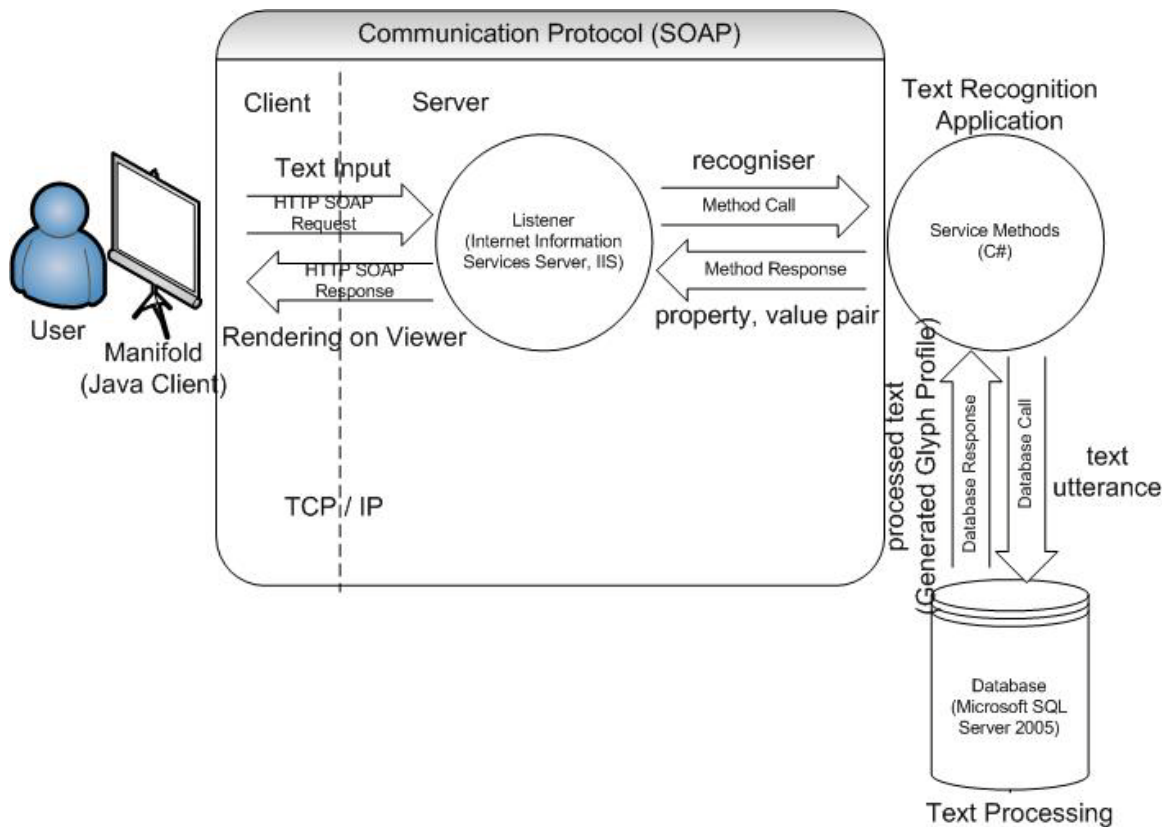


Figure 57: Manifold interaction with the remote server through SOAP web service

Custom glyph provides a convenient way of drawing graphical objects for the users who prefer to obtain results in single step, rather than going through various stages, like modifying different glyph properties independently through various property editors.

Observation 6.1: *It is noteworthy here that the process of applying transforms in succession is “cumulative”, but not “commutative”. In other words, applying translation and rotation, for example, causes the output graphic to be translated and also rotated. However, depending on your parameters, you might not get the same results if you switch the order of translating and rotating. Care must be taken when applying these transforms on the glyphs in succession.*

Design Issue 6.5: *The current implementation of Manifold as a two-dimensional graphical editor doesn't provide property editors for editing glyph properties like translation, scaling, and rotation. The future versions should implement these. When implemented, the usefulness of the custom glyph menu item could be argued in terms of gain in speed or convenience. However, I would argue that it would still remain an easy resort for performing multiple tasks in a single step, especially for the users who prefer typing.*

Design Issue 6.6: *If the Manifold viewer is zoomed in or out, and the user wants to create a glyph using the custom glyph menu item, the viewer gets reset to its original view for rendering the new glyph. A possible alternative could be to draw the new glyph on the zoomed workspace without altering the zoom effect, previously executed by the user.*

6.6.4 Smart Art Menu Item

We define a smart-art as a collection of glyphs created by drawing glyphs separately and then grouping them to make a single art, like a smiley face. Different objects in the art could be manipulated individually at any point by un-grouping (Chapter 5, section 5.4.3) them and then grouping them again to form the modified art. Thus, the *Smart Art* menu item is used to insert the previously saved poly-glyphs, providing an easy and effective way to manipulate grouped glyphs at a later stage without the need to draw them from scratch again.

The working of Smart Art menu item is similar to that of the Rectangle, Ellipse and Line menu items as discussed in section 6.6.1.1, 6.6.1.2, and 6.6.1.3 respectively, with the only difference being that the `CLIP_ART_TYPE` is set to “grouper”. This menu item is implemented in class `ClipArtSmartArt.java`. When the class `TableSavedGlyphs.java` is invoked by passing the `CLIP_ART_TYPE` as “grouper”, it draws child glyphs, one at a time, on the Manifold viewer and adds them to the parent glyph to be rendered as a smart-art (a grouped/composite object) on Manifold. This is done by the method `renderChildGlyphOnViewer(int)` that draws all the glyphs that are child to the grouper glyph (the parent), which are set by calling the method `setGrouperChildren(int parentId)`. The method calls the stored procedure `GET_CLIP_ART` that takes two parameter, `clipArtType` and `parentGlyphId`. The `clipArtType` here is “grouper” as discussed above and the `parentGlyphId` is the id (the field from the SQL table and not the `nodeId` from Manifold) of the grouper glyph created. This stored procedure returns all the child glyph profiles for the current parent grouper glyph, which are added to the parent by the method `renderChildGlyphOnViewer(int)`.

Chapter 7

Property Editors

As discussed in Chapter 1, the design of Manifold contains Property Viewer which is used to display editable properties (attributes) of the glyphs in the workspace in the form of property editors. In the earlier versions of Manifold, Property Viewer displays only the editable properties of single selected glyph at a time. In Chapter 4, I described, how I changed this to display the common editable properties of multiple selected glyphs that would allow modifying the properties of multiple glyphs at a time. In other words, the property viewer again queries all the selected leaf glyphs to determine the common properties. Such complex features may look fancy, but the important question is what value they present to the user (Chapter 4).

In this chapter I will discuss the addition of new property editors with respect to the new glyphs (as discussed in Chapter 5), and few modifications to the old property editors as well.

7.1 Introduction

When computer needs specific information, it initiates a *dialog* with the user. “Dialog”, as understood in graphical user interface design, is a conversational vignette in a limited domain, on a well defined topic, with a goal of extracting specific information from the user. The computer knows in advance what to ask and the full range of options it can expect from the user as an answer.

Property Viewer displays the properties for editing of the selected glyphs. Each property has a different editor, depending on the property's data type, as illustrated in Figure 58. 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 becomes the currently selected glyph.

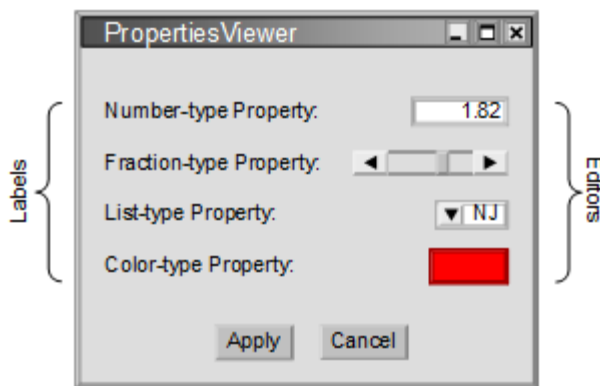


Figure 58: Example of a property editing dialog box. Property editors allow editing the property values. [4]

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.

The composition of the current `manifold.swing.PropertiesViewer` implementation is shown in Figure 59. The glyph-specific property editors are contained in the `PropertyEditorsPanel` (package `manifold.swing`), 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` [27].

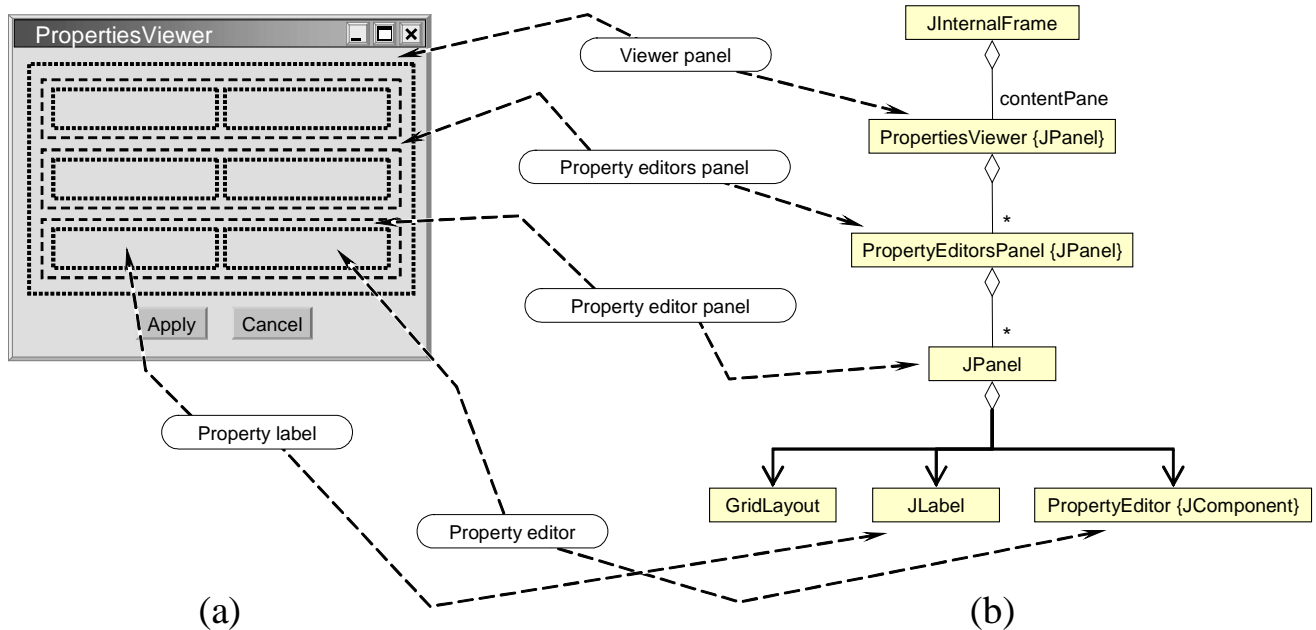


Figure 59: Composition of a property editor dialog box: (a) The screen rendering; (b) The UML class diagram [4]

The `PropertyEditorsPanel` is a subclass of `javax.swing.JPanel` [28]. Within the `PropertyEditorsPanel` are a set of smaller `JPanel`'s with a *Grid Layout* of 1x2 as shown in Figure 59. Each of this subset (1x2) `JPanel`s hold the editable properties of a selected glyph. The left box (i.e. {1, 1}) contains a `javax.swing.JLabel` [29] 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.

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 `Hashtable` [30] 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 type.

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. Furthermore, it *de-couples* the `PropertyEditorsPanel` to obtain the `JComponent` (i.e. the editable properties) and assigns them to a generic interface `manifold.PropertyEditor`. Through the `PropertyEditor` interface the properties are subsequently altered.

7.2 Design: Property Editor

Property editors are used to edit the properties/attributes of a selected glyph. Each of these properties is represented by a `JComponent` [27]. These components belong to the package `manifold.swing.editors`, where each of these editors is implemented in a different class. These classes implements the `PropertyEditor` interface (package `manifold`), `ActionListener` [31]/`ChangeListener`[32]; and extend a `javax.swing` object like `JPanel`, `JButton` [34], etc.

An editor can belong to multiple glyphs. In order to tell the application what property editors should be rendered on the Property Viewer, an XML file “editors.xml” (provided with the `Manifold` package) is configured accordingly. This allows sharing of a single property editor with multiple glyphs, by configuring just the XML file. For

example, consider the code stub below for the glyph of type “rectangle”, defined in the “editors.xml”, to render various property editors.

```
<!-- ***** Rectangle Editor Panel ***** -->
<void method="put">
  <string>rectangle</string>
  <object class="manifold.swing.PropertyEditorsPanel">
    <void method="add">
      <object
        class="manifold.swing.editors.FillColorEditor">
        <void property="propertyName">
          <string>fill.color</string>
        </void>
      </object>
    </void>
  </object>
</void>
```

Here, The Fill Color Editor is defined for the glyph of type “rectangle”, so that whenever a rectangle glyph type is drawn on the Manifold viewer, this editor gets rendered on the Property Viewer.

7.3 New Property Editors

I worked on the implementation of the following new property editors:

- *Text Editor*: allows changing text of the *Text* glyph type.
- *Font Editor*: allows changing font of the *Text* glyph type.
- *Image Editor*: allows importing a new image for the *Image* glyph type.

7.3.1 Text Editor

Text Editor provides the user an option to edit the text of a *Text* glyph type. This editor is implemented in the class `TextEditor.java` belonging to package

manifold.swing.editors. This class extends to a JPanel that acts as a placeholder for a JTextField [33], and implements manifold.PropertyEditor and java.awt.event.ActionListener to read the change in the text when the return (enter) key is pressed from the keyboard. The action event is read and processed in the method actionPerformed(ActionEvent event_), that reads the new text when return key is pressed from the JTextField that registers the object created using the addActionListener() method in the constructor of the class. The code stub for the actionPerformed() method is defined as:

```
textFieldText_ = textBox.getText();//Read text in a String field

// Make an event frame to request the application domain
// for property change.
Hashtable slots_ = null;
for (int i = 0; i < currentNodeId.length; i++) {
    slots_ = new Hashtable();
    slots_.put(EventFrame.VERB, ControllerImpl.SET_PROPERTIES);
    slots_.put(EventFrame.NODE_ID, currentNodeId[i]);
    slots_.put(propertyName, textFieldText_);
    propertiesViewer.getController().sendAsyncEvent(new
    EventFrame(slots_));
}
```

In the above code stub, the text entered in the JTextField (variable name, textBox), is read in a String variable textFieldText_. The following steps generate an EventFrame (package manifold) 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* (text.text) and the entered text are sent to the controller via the EventFrame. The values specified in the EventFrame are stored in the cachedState hash table if they do not exist or are updated if they exist. Addition of

the text editor and rendering the new text of the glyph required some changes in the `draw()` method of the class `Text.java` (package `manifold.impl2D.glyphs`), which were explained in the Chapter 5, section 5.3.1.

The default text of the *Text* glyph is provided in the file “tools.xml”, distributed with the manifold source packages. A code stub from the file that renders the default text when the Text Glyph is drawn for the first time is shown below:

```
<void method="put">
    <string>text.text</string>
    <string>Your Text Here</string>
</void>
```

Figure 60 shows the working of the *Text Editor* that allows user to change the text of the *Text* glyph.

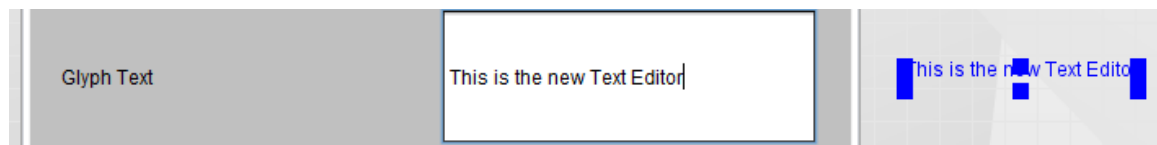


Figure 60: The Text Editor. When the user presses the return (enter) key, the new text is rendered on the Text glyph.

7.3.2 Font Editor

In order to provide the *Text* glyph with more features like changing the font properties (the style, the size and the face), a new editor called *Font Editor* was implemented. This feature was important to allow decorating the text of the Text Glyph. The three main important properties of a text’s font are:

- The choice of Font type (font face) like “Courier”, “Arial”, “Times New Roman”, etc.

- The choice of font style like “**bold**”, “*italic*”.
- The choice of “increasing” or “decreasing” the font size.

The idea behind implementing the font editor was to allow changing these three important properties of text font belonging to the *Text* glyph. It should be noted that there could also be other font properties like *underlining*, *shading*, etc. which are not implemented in the current Manifold version.

The class `manifold.swing.editors.FontEditor` defines the code necessary to implement Font Editor. The class implements three interfaces `manifold.PropertyEditor`, `java.awt.event.ActionListener`, and `javax.swing.event.ChangeListener`; and extends `javax.swing.JPanel`. The `JPanel` is used to hold three `JComponents`, two `JComboBox` [35] and one `JSpinner` [36]. A `JComboBox` 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, one of the `JComboBox` conatins a list of font style names ("Plain", "Bold", "Italic" and “Bold Italic”). The other `JComboBox` contains a list of font faces, which are read from the `java.awt.GraphicsEnvironment` [37]. A `JSpinner` provides a single line input field that lets the user select a number or an object value from an ordered sequence. Spinners typically provide a pair of tiny arrow buttons for stepping through the elements of the sequence (the keyboard up/down arrow keys also cycle through the elements). The user may also be allowed to type a (legal) value directly into the spinner. Here, the `JSpinner` is used to allow changing the size of the font using a `javax.swing.SpinnerNumberModel` [38].

The two JComboBox are added to the ActionListener via the addActionListener() method, and the JSpinner is added to the ChangeListener via the method addChangeListener(), in the constructor of the FontEditor.java class. This is because the application has to process an action event when a user clicks the JComboBox and a change event when the user clicks on the JSpinner. When an action event occurs, the objects actionPerformed() method is invoked that has the following code stub:

```
fontChoice = (String)fontsComboBox.getSelectedItem();
styleChoice = stylesComboBox.getSelectedIndex();

font_ = new Font(fontChoice, styleChoice, sizeChoice);

// Make an event frame to request the application domain
// for property change.
Hashtable slots_ = null;
for(int i=0;i<currentNodeId.length;i++) {
    slots_ = new Hashtable();
    slots_.put(EventFrame.VERB, ControllerImpl.SET_PROPERTIES);
    slots_.put(EventFrame.NODE_ID, currentNodeId[i]);
    slots_.put(propertyName, font_);
    propertiesViewer.getController().sendAsyncEvent(new
    EventFrame(slots_));
}
```

Here, the font face and the font style are read from the two JComboBox, and the new font is added to the EventFrame, as discussed in the previous section.

Similarly, when a change event occurs, the objects stateChanged() method is invoked that has the following code stub:

```
try {
    String size = sizesSpinner.getModel().getValue().toString();
    sizeChoice = Integer.parseInt(size);

    font_ = new Font(fontChoice, styleChoice, sizeChoice);

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

```

Hashtable slots_ = null;
for (int i = 0; i < currentNodeId.length; i++) {
    slots_ = new Hashtable();
    slots_.put(EventFrame.VERB, ControllerImpl.SET_PROPERTIES);
    slots_.put(EventFrame.NODE_ID, currentNodeId[i]);
    slots_.put(propertyName, font_);
    propertiesViewer.getController().sendAsyncEvent(new
        EventFrame(slots_));
}
} catch (NumberFormatException nfe) {
    System.out.println(nfe.toString());
}

```

Here, the new font size (an integer) is read from the JSpinner, and the new font is added to the EventFrame. The method is in a try-catch block, to catch an exception of type `NumberFormatException` [39].

Design Issue 7.1: *A new Hashtable object is created for editing properties of each selected glyph. This might lead to performance issues. A better approach to handle this must be thought about. One such approach was discussed in [78], which claimed performance improvement by eliminating the use of Hashtable's and using comma-separated string fields to hold the property-value pairs. Another work-around could be, developing a new EventFrame type that could handle multiple glyphs properties within a single HashTable.*

Similar to the *Text Editor* (as discussed in section 7.3.1), rendering the new font properties of the glyph required some changes in the `draw()` method of the class `Text.java` (package `manifold.impl2D.glyphs`), which was explained in Chapter 5, section 5.3.1.

The default font of the Text Glyph is provided in the file “tools.xml” (distributed with Manifold source packages). A code stub from the file that renders the default font when the Text Glyph is drawn for the first time is shown below:

```
<void method="put">
  <string>text.font</string>
  <object class="java.awt.Font">
    <string>Dialog</string>
    <int>0</int>
    <int>12</int>
  </object>
</void>
```



Figure 61: The Font Editor. Notice the new font size and style that gets rendered on the Text glyph.

Figure 61 shows the working of the Font Editor that allows user to change the font properties of the *Text* glyph type.

Design Issue 7.2: *The text and font editors are implemented separately rather than being a part of the same editor modifying different properties of the Text glyph type. This design choice was made mainly to keep editors for different glyph properties (text and font here) separate from each other which also provided implementation clarity at the class and code level.*

7.3.3 Image Editor

The *Image* glyph was discussed in Chapter 5, section 5.4.1, where the user can render an image on the Manifold viewer space. Two methods of rendering the image

were also discussed one from the *File Browser*, and the other from the database. The *Image Editor* allows importing new images, selected from the file system, for the *Image* glyph. However, it doesn't allow changing the image pixels, such as in Photoshop [10].

The class `manifold.swing.editors.ImageEditor` defines the code necessary to implement the Image Editor. The class implements two interfaces `manifold.PropertyEditor` and `java.awt.event.ActionListener`; and extends `javax.swing.JButton`. The `JButton` is added to the `ActionListener` via the `addActionListener()` method in the constructor of the class `ImageEditor.java`. This is because the application has to process an action event when a user clicks the `JButton`. When an action event occurs, the objects `actionPerformed()` method is invoked that has the following code stub:

```
JFileChooser chooser = new JFileChooser();

int returnVal = chooser.showDialog(this, "Select an Image");

if(returnVal == JFileChooser.APPROVE_OPTION) {
    File file_ = chooser.getSelectedFile();
    // Make an event frame to request the application domain
    // for property change.
    Hashtable slots_ = null;
    for(int i=0;i<currentNodeId.length;i++) {
        slots_ = new Hashtable();
        slots_.put(EventFrame.VERB, ControllerImpl.SET_PROPERTIES);
        slots_.put(EventFrame.NODE_ID, currentNodeId[i]);
        slots_.put(propertyName, file_);
        propertiesViewer.getController().sendAsyncEvent(new
            EventFrame(slots_));
    }
}
```

Here, a `JFileChooser` [40] dialog allows the user to select a new image from the file system, whose value is read and passed on to the application domain for a

property change request, and subsequently the new image is rendered in the `draw()` method of the class `manifold.impl2D.glyphs.Picture.java` that was discussed in Chapter 5, section 5.4.1

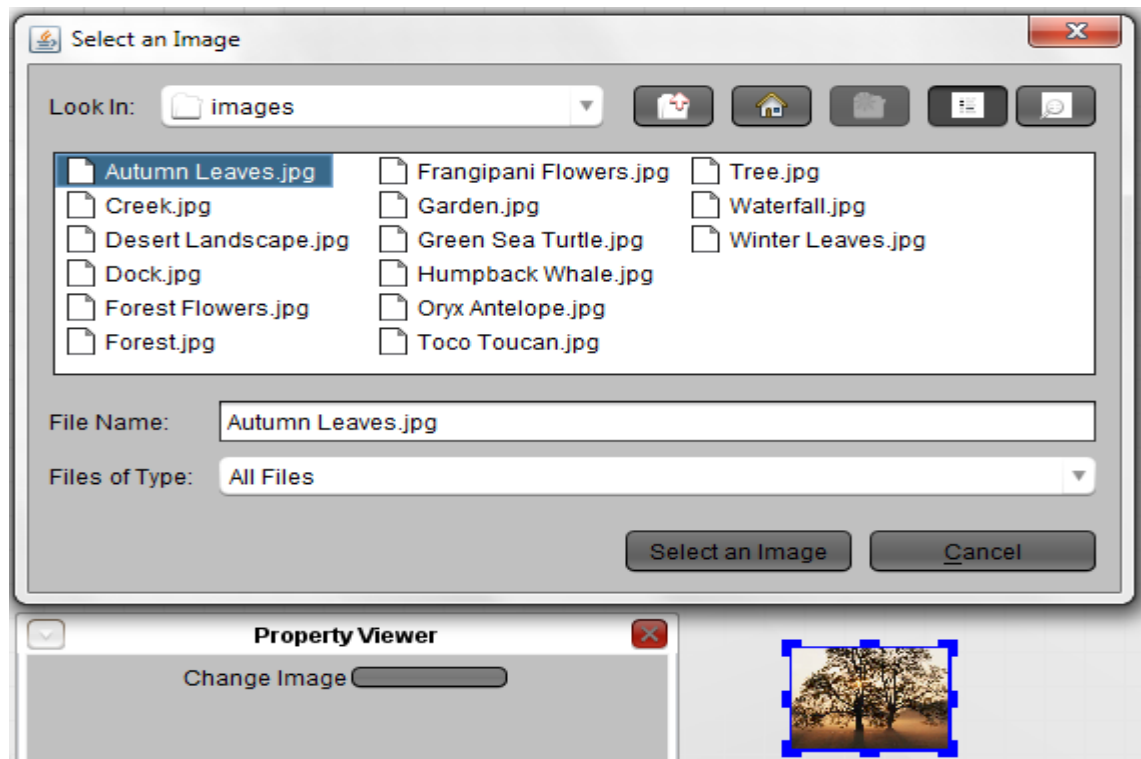


Figure 62: Image Editor: When the user clicks on the button, a file browser appears where the user can select the new Image.

Figure 62 shows the working of the Image Editor that allows user to change the image of the Picture/Image glyph type. Figure 63 shows the new image that gets rendered as an action output of user.

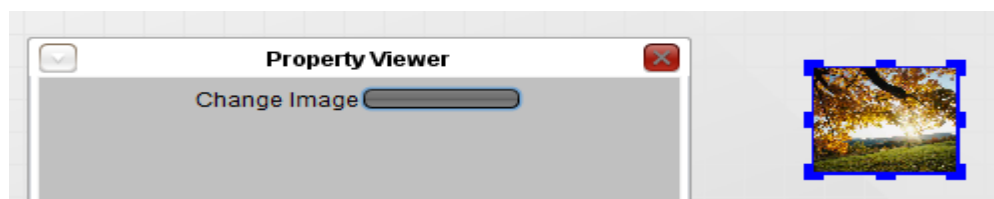


Figure 63: New image being rendered as a result of user's new selection from the file browser.

Chapter 8

Complexity and Performance

Code is complex, but it doesn't have to be overly complex. Keeping the code and design to manageable levels is one of the key challenges in software development today. However, code complexity alone cannot affect the performance of software. If a complex code is well sequestered and has an efficient architecture (the interdependency among various software components), it could be effectively managed and lead to improved performance.

This chapter discusses the design and structural complexity of the current Manifold version and then studies the performance of the newly incorporated features.

8.1 Design Complexity

The designs usually start elegant, but as the number of classes grows, the complexity inevitably creeps in. It is an important issue about any software product. There are a few quantitative measures of software complexity [69, 70], and this section discusses them for Manifold, in terms of its stability, quantitative and deterministic evaluation of its structure, its dependency structure (helpful in refactoring components). Also, I will discuss some of the architectural problems inherent in Manifold's design.

8.1.1 Structural Analysis

Figure 64 shows the core structure of Manifold. Manifold has an “onion” structure (as shown in Figure 65), where the developer can start using at any layer of the onion.

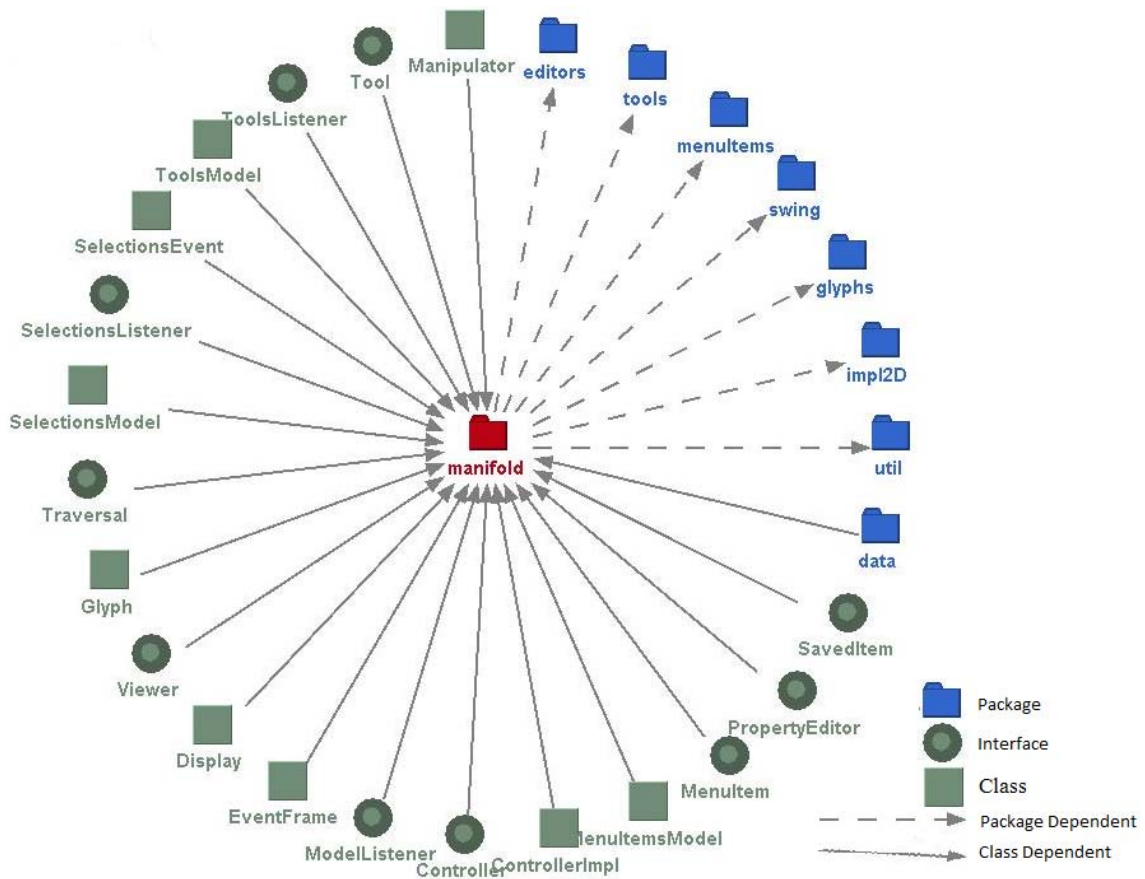


Figure 64: Manifold Core Structure

If you choose to start with the core interfaces only (the package manifold), you are only using the high level design. The package manifold.impl2D offers basic two-dimensional geometry functions. Instead of this, you could build on top of the core interfaces and implement an equivalent package to be used on small devices. The

package `manifold.swing` offers layout and controls implemented using the Java Swing GUI toolkit. Package `manifold.data` provides the semantics to connect to a database, and could be modified to attach any database system with Manifold. Figure 66 shows this relationship with package `manifold` being core and different levels (and areas) of implementation on it.

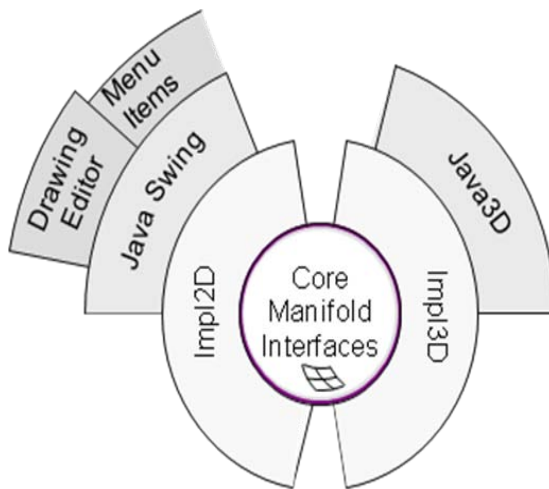


Figure 65: The “onion” structure of Manifold packages

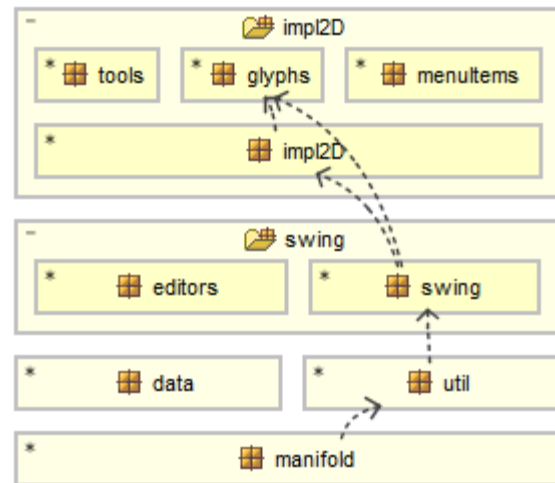


Figure 66: Package level Architecture of Manifold (The arrows specifies use, for e.g. manifold uses util)

The current implementation offers a simple two-dimensional drawing editor with a constant effort of having minimum coupling at class and package level (refer to section 8.1.2). However, there were certain relationships that were inherent for the implementation of Manifold (*core packages*). The following sections would discuss some of the structural and code analysis of Manifold.

Manifold currently has a total of 127 *objects* (classes, interfaces and packages), forming a total of 809 *relationships* (not including Java library classes) i.e. extends, implements, uses (method argument or returned from the method), and contains

relationships. A typical object in this system immediately depends on 6.37 objects. On average, the modification of one object potentially affects 16.3 other objects. The overall stability [81] of the system is calculated to be 87% (calculated as a function of the average number of affected objects). 100% stability specifies that all the objects are completely decoupled from other objects.

$$\text{Stability} = (1 - \text{average number of affected objects} / \text{total number of objects}) * 100$$

$$\Rightarrow (1 - 16.3/127) * 100 = 87\% \text{ (for Manifold)}$$

The key statistics of Manifold has been specified in Table 9. The calculations were based on formulas and metrics specified in [81].

Table 9: Current Manifold Structure Statistics Summary

Property	Value
Number of Objects (classes, interfaces and packages)	127
Number of packages	13
Number of Relationships i.e. extends, implements, uses (method argument or returned from the method), and contains relationships.	809
Maximum Dependencies (classes that depends on one class)	31
Minimum Dependencies (classes that depends on one class)	0
Average Dependencies (classes on which this class depends on)	6.37
Maximum Dependents (classes on which this class depends on)	71
Minimum Dependents (classes on which this class depends on)	0
Average Dependents (classes on which this class depends on)	6.37
Relationship to Object Ratio	6.37
Number of objects affected by modification of one object	16.3

The chart in Figure 67 plots the number of classes having different counts of dependencies (input links) and dependents (output links) to other Manifold classes. Only the connections to Manifold classes are shown; the connections to Java classes are not shown. As expected, most of the classes have very few links and vice versa: very few

classes have many links. This characteristic was observed for very large software packages, as well [86].

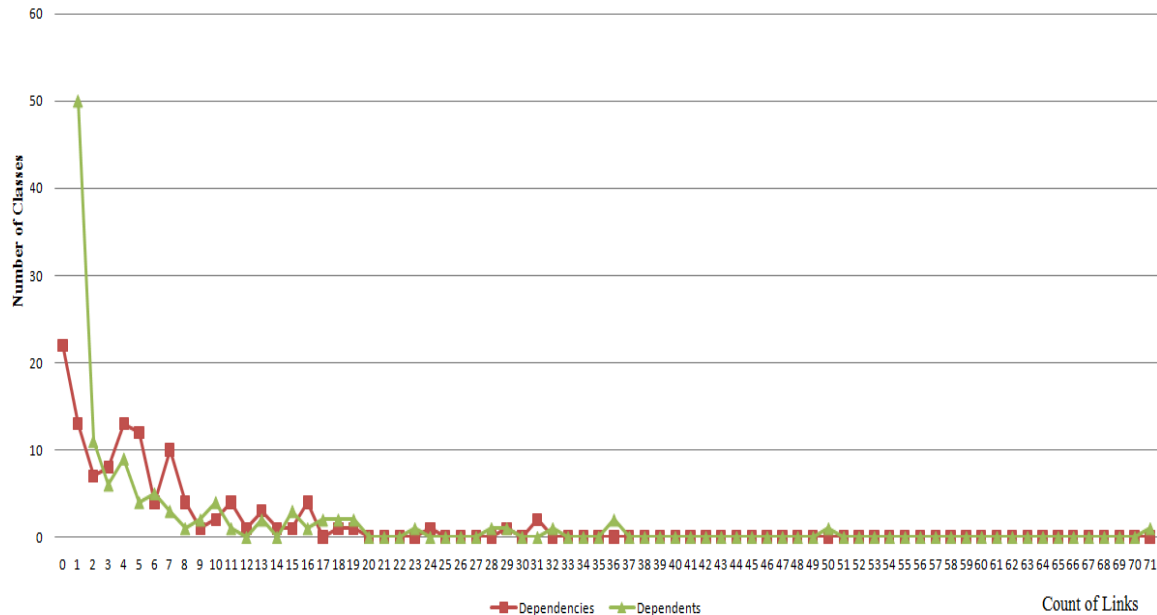


Figure 67: Statistics of the connectivity of all the classes in the current Manifold implementation.

The classes with maximum dependencies are `MapOverlayGlyphViewer` and `Viewer2DImpl` (both belonging to package `manifold.swing`), as these mainly act as the background viewer of the Manifold application and are responsible for rendering all the graphics on the canvas. The class with the maximum dependents is `Viewer` (package `manifold`) as it is used by all the classes that require any screen rendering. Obviously, these are central classes, so it is to be expected that they should have greater connectivity than any others.

The current design is arguably lightweight and simple even with a feature-laden interface than the previous versions. See [4], section 8.1 for a similar comparison for an earlier, lighter and simpler Manifold version.

8.1.2 Excessive Structural Complexity

The excessive structural complexity [70] is a set of thresholds for “Fat” and “Tangles” at different levels of composition. *Fat* indicates “too much stuff” in one place and is measurable at every level of composition viz. design, package, class and method. Cyclomatic Complexity [69] is used to measure fat at the method level and the number of dependencies in the child dependency graph is used at other levels. *Tangles* define the cyclic dependencies between packages. Both Fat and Tangles are measured as a percentage by which the given threshold is exceeded.

Using the above definitions and metrics and threshold levels, as defined in [83], we performed the structural analysis of Manifold and the following results were obtained.

- **Design Level Tangles** (High level packages that contains sub-packages),

Threshold = 0 [83]:

- manifold.impl2D (8%)
- manifold (7%)

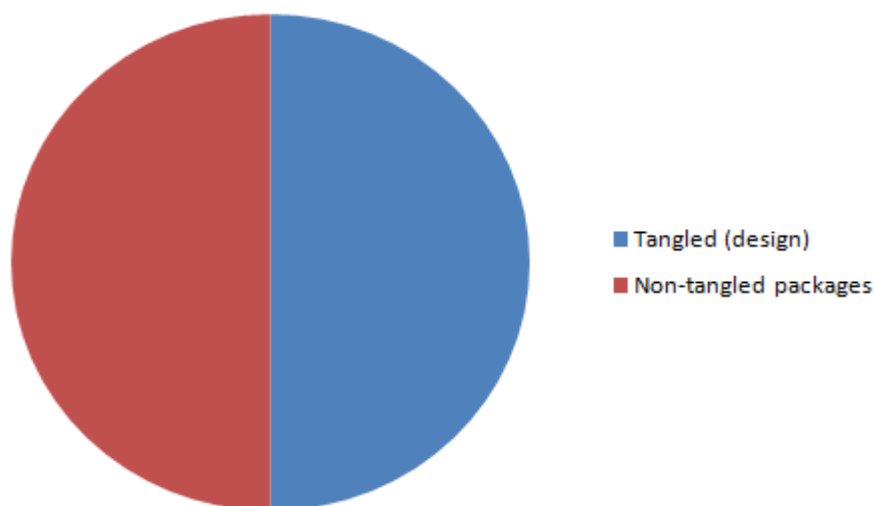


Figure 68: Pie Chart showing Tangles in Manifold design

This is shown in Figure 68. This value is calculated as the number of dependencies that go bottom-up (interdependencies) as a percentage of all the dependencies on the child graph of the package. These two packages represent the core of Manifold (*model* and *controller*), and hence there is a level of communication that is inherent for their functioning. This contributes 89% to Excessive Structural Complexity.

- **Total Fat, design level**, Threshold = 120 [83]: No items exceed the threshold for Fat at the design level (0 of 4). The value is calculated as the number of dependencies in the child graph of the item, i.e. the dependency graph of the sub-packages of the measured package.
- **Total Fat, leaf package**, Threshold = 120 [83]: No items (leaf packages) exceed the threshold for Fat at the leaf package level (0 of 13). The value is calculated as the number of dependencies in the child graph of the item, i.e. the dependency graph of the classes in the measured package.
- **Total Fat, class level**, Threshold = 120 [83]: No items (classes) exceed the threshold for Fat at the class level (0 of 127 total classes). The value is calculated as the number of dependencies in the child graph of the item, i.e. the dependency graph of the methods and fields in the class.
- **Total Fat, methods**, Threshold = 15 [83]: (2 of 980 total methods)
 - manifold.swing.TableSavedGlyphs.createGlyphProperties(Hashtable, AffineTransform, Object[][], int, String):Hashtable


```

o manifold.swing.TableSavedDocuments.createGlyphPro
  perties(Hashtable, AffineTransform, Object[][],
    int, String):Hashtable

```

Figure 69 distribution of fat vs. non-fat methods in Manifold source packages. The value is calculated as the Cyclomatic Complexity [69] (McCabe's metric) of the method (see section 8.1.3 for details on why these methods are fat). This contributes 11% to Excessive Structural Complexity.

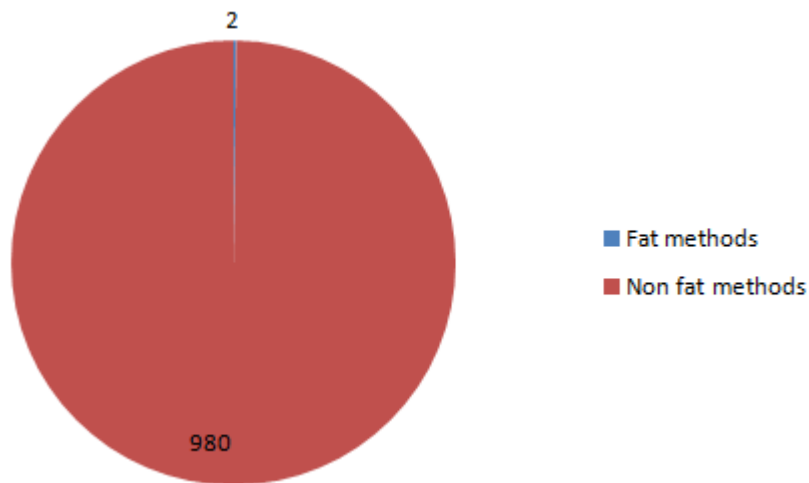


Figure 69: Fat vs. non-Fat methods in Manifold

The entire structural level complexity is summarized in the pie- chart shown in Figure 70 which shows that the main contributors to structural complexity are 2 high-level packages (manifold and manifold.impl2D) and 2 methods viz. manifold.swing.TableSavedGlyphs.createGlyphProperties and manifold.swing.TableSavedDocuments.createGlyphProperties.

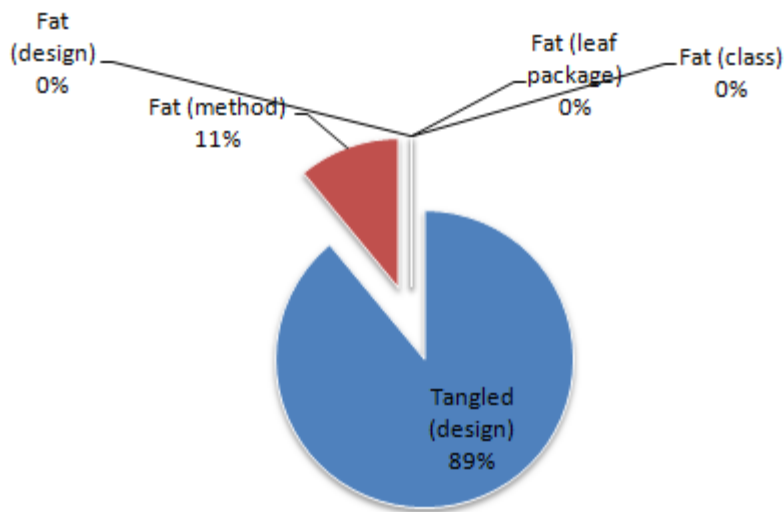


Figure 70: Pie Chart showing the contributors to the Structural Level Complexity of Manifold

8.1.3 Code Analysis

As discussed above, if the structure (design level fat and tangles) of an application is better, then it can overshadow complex code (difficult re-usability). However, there is a broad line between complex code and fat code (high cyclomatic complexity). The complexity of code essentially determines its reuse (by the same or different programmers) and testing. Though these seem trivial, most of the time such jobs can become non-trivial due to the complexity of the code. One way to overcome this is to provide sufficient documentation with the code, so that it becomes easy to understand at a later stage. This however, would make the code somewhat easy to understand, and not less complex. It thus becomes essential to analyze the code complexity in terms of various pre-defined metrics.

Code complexity metrics are directly related to the maintainability and testability of the code [84]. The more complex a code is the less maintainable and testable it is. If

the code is object oriented, the complexity metrics also have a direct bearing on the extensibility and modularity of the code. Maintenance metrics can be subdivided into formatting metrics and logical metrics [84]. *Formatting metrics* deals with aspects such as indentation conventions, code comment guidelines, naming conventions, white space usage etc. *Logical metrics* give you information about aspects directly associated with program logic and code flow - these are things such as the number of paths through a program, depth of conditional statements and blocks, the number of parameters to functions etc. One such logical metric is the cyclomatic complexity of the code.

Cyclomatic complexity [69] gives the number of paths that may be taken when a program is executed. Methods with a high cyclomatic complexity tend to be more difficult to understand and maintain. Some of the tokens (in Java) responsible for the program taking different paths during execution are [85]:

- while & do while statements.
- if statements.
- for statements.
- Ternary Operators & Logical Operators.
- switch case statements.
- return, throw, throws, catch statement.

Before analyzing the cyclomatic complexity of various Manifold classes, let us look at summary of Manifold classes and packages in terms of the top four classes and packages having the highest number of lines of codes. This is shown in Table 10.

Table 10: Top 5 Manifold classes and packages in terms of LOC

Classes Lines of Code	
Class Name	Lines of Code
Selector	646
TableSavedGlyphs	608
TableSavedDocuments	527
Viewer2DImpl	516
Package Lines of Code	
Package Name	Lines of Code
manifold	21364
manifold.impl2D	9253
manifold.swing	8348
manifold.tools	2892
Total (Non-comment Non Blank LOC)	~11K

Table 11 defines metrics for dividing methods into three levels according to their cyclomatic complexity [82].

Table 11: Code Complexity Metrics

Metric	CC Count	Legend
High Complexity	7	Red
Moderate Complexity	4	Yellow
Low Complexity	0	Green
Interfaces	N/A	Grey (Interface)
Classes	N/A	Vertical Bars (Horizontal Axis)
Methods	N/A	Rectangular Blocks (Vertical Axis)

Now, using the information from Table 11, let us consider different Manifold packages (figures 56-63) and look into the methods with high complexity and the reason for the same. These results were obtained using open-source software package CyVis [82].

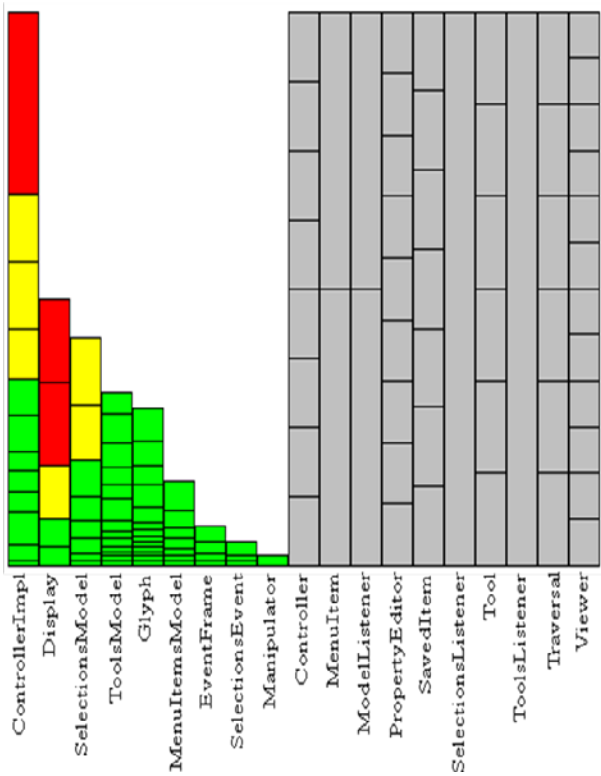


Figure 71: Methods with different levels of complexity in package manifold

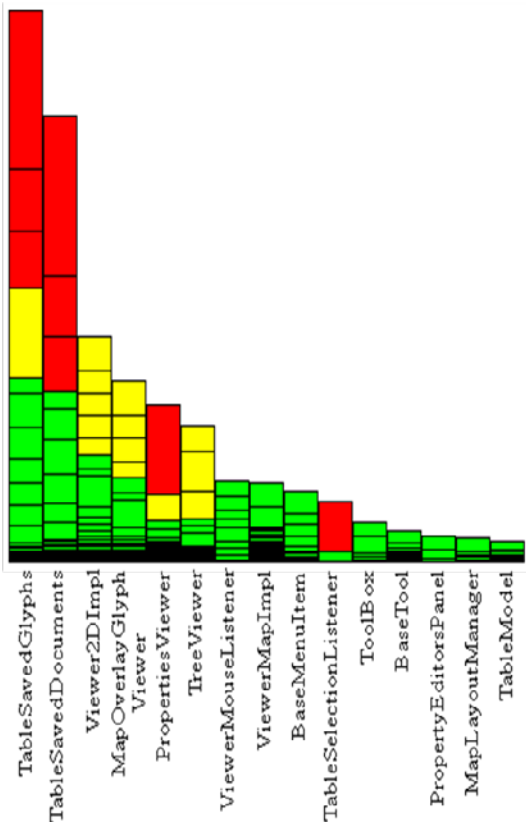


Figure 72: Methods with different levels of complexity in package manifold.swing

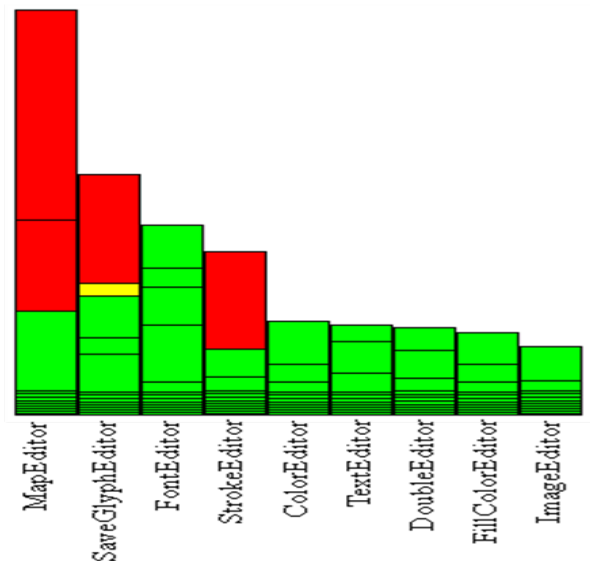


Figure 73: Methods with different levels of complexity in package manifold.swing.editors

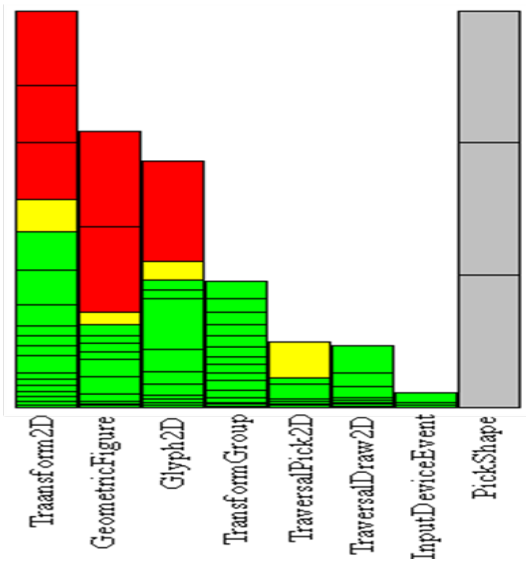


Figure 74: Methods with different levels of complexity in package manifold.impl2D

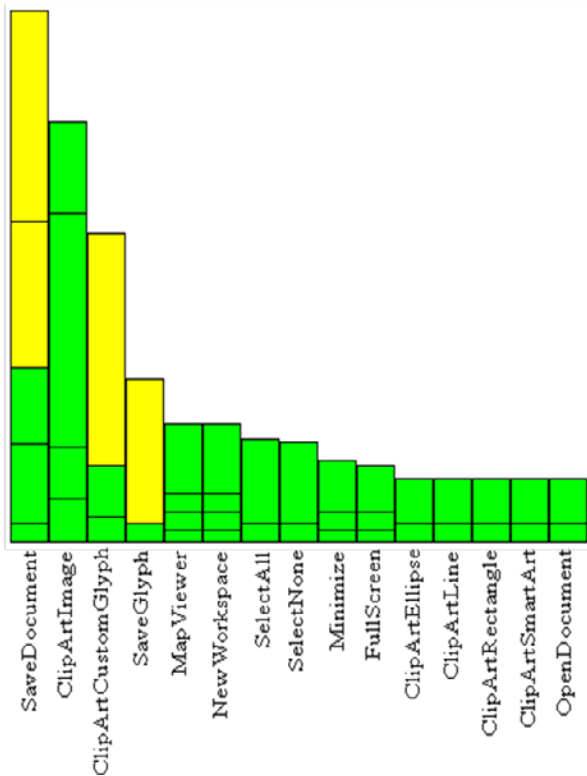


Figure 75: Methods with different levels of complexity in package manifold.impl2D.menuitems

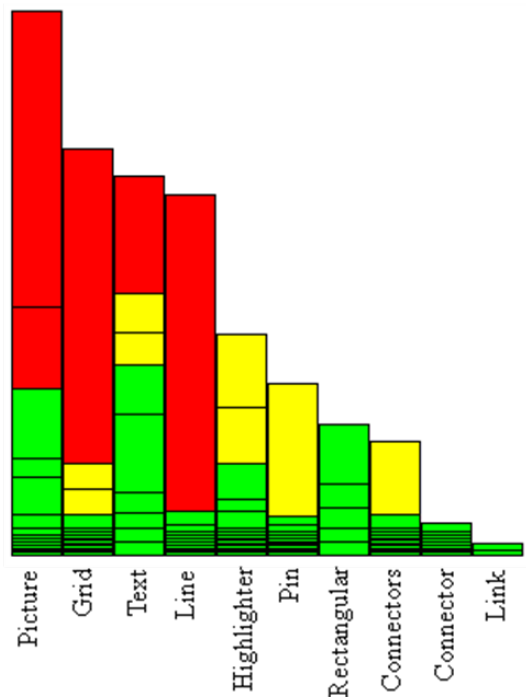


Figure 76: Methods with different levels of complexity in package manifold.impl2D.glyphs

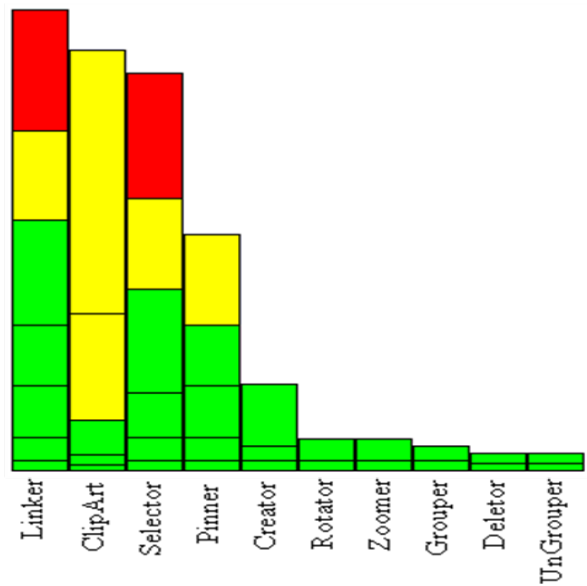


Figure 77: Methods with different levels of complexity in package manifold.impl2D.tools



Figure 78: Methods with different levels of complexity in package manifold.data

- **Package:** manifold

Figure 71 shows classes and methods in package manifold.

Table 12 shows the methods with high cyclomatic complexity (CC) in package manifold.

Table 12: High Code Complexity methods in package manifold

Class Name	Method Name	CC	Reason
ControllerImpl	sendAsyncEvent	9	If-else blocks to handle different event frames and report errors of any unaccounted frame.
Display	run	9	Provides synchronized access to viewers and runs forever as a separate thread to control the frame-rate.
Display	removeViewer	8	Synchronized method to remove a given viewer from receiving periodic notifications.

The high complexity of `sendAsyncEvent` (`ControllerImpl`) is inherent due to the fact that for handling ‘n’ event frames we need to have ‘n’ conditional blocks.

- **Package:** manifold.swing

Figure 72 shows classes and methods in package manifold.swing.

Table 13 shows the methods with high cyclomatic complexity in package manifold.swing.

The high complexity of method `createGlyphProperties()` is inherent due to the fact that for handling ‘n’ property-value pairs, we need to have ‘n’ conditional blocks.

Table 13: High Code Complexity methods in package manifold.swing

Class Name	Method Name	CC	Reason
TableSavedGlyphs	createGlyphProperties	26	If-else blocks to create a glyph's property-value pairs as a Hashtable.
TableSavedGlyphs	setGrouperChildren	9	Try-catch block to read data from database.
TableSavedGlyphs	getSavedItemFromDatabase	9	Try-catch block to read data from database.
TableSavedDocuments	createGlyphProperties	26	If-else blocks to create a glyph's property-value pairs as a Hashtable.
TableSavedDocuments	getAllGlyphsInDocument	9	Try-catch block to read data from database.
TableSavedDocuments	getSavedItemFromDatabase	9	Try-catch block to read data from database.

- **Package:** manifold.swing.editors

Figure 73 shows classes and methods in package manifold.swing.editors.

Table 14 shows the methods with high cyclomatic complexity in package manifold.swing.editors.

Table 14: High Code Complexity methods in package manifold.swing.editors

Class Name	Method Name	Cyclomatic Complexity	Reason
StrokeEditor	actionPerformed	9	If-else blocks to deal with 6 different types of strokes.

- **Package:** manifold.impl2D

Figure 74 shows classes and methods in package manifold.impl2D.

Table 15 shows the methods with high cyclomatic complexity in package manifold.impl2D.

Table 15: High Code Complexity methods in package manifold.impl2D

Class Name	Method Name	CC	Reason
Transform2D	constructor	8	Nested if-else blocks to compose component transformations.
Transform2D	extractComponents5	7	Geometric calculations for decomposing the given transformation into the constituent components of "pure" translation/rotation/scale.
Transform2D	createInverse	7	Overrides AffineTransform method createInverse() to handle zero-scaling and infinite values, and throws exception.
GeometricFigure	draw	9	Geometric calculations from 6-element to 5-element transform matrix.
GeometricFigure	saveGlyph	9	Interacts with database to save a glyph using try-catch blocks.

- **Package:** manifold.impl2D.menuItems

Figure 75 shows classes and methods in package manifold.impl2D.menuItems.

This package does not contain any methods with high cyclomatic complexity.

- **Package:** manifold.impl2D.glyphs

Figure 76 shows classes and methods in package manifold.impl2D.glyphs. Table 16 shows the methods with high cyclomatic complexity in package manifold.impl2D.glyphs.

Table 16: High Code Complexity methods in package manifold.impl2D.glyphs

Class Name	Method Name	CC	Reason
Picture	draw	8	Large method for image rendering and controls its geometric interpretation on Manifold (more lines, and if-else blocks).
Picture	getImagesFromDb	7	Interact with database in try-catch blocks to read names of images persisted.
Grid	draw	12	Large method for drawing Grid (background of workspace) and controls its geometric interpretation on Manifold (more lines, and if-else blocks).
Line	translateAndScaleShape	9	If-else blocks to control geometric translation and shape of Line glyph type.
Text	draw	9	Large method for rendering text glyph and controls its geometric interpretation on Manifold (more lines, and if-else blocks).

- **Package:** manifold.impl2D.tools

Figure 77 shows classes and methods in package manifold.impl2D.tools.

Table 17 shows the methods with high cyclomatic complexity in package manifold.impl2D.tools.

Table 17: High Code Complexity methods in package manifold.impl2D.tools

Class Name	Method Name	CC	Reason
Linker	drawSelectionBox	7	Ternary and Logical operators for drawing rubber-band object for multiple glyph selections.
Selector	drawSelectionBox	7	Ternary and Logical operators for drawing rubber-band object for multiple glyph selections.

- **Package:** manifold.data

Figure 78 shows classes and methods in package manifold.data. This package does not contain any methods with high cyclomatic complexity.

Having discussed different methods in Manifold with high cyclomatic complexity, I would like to point out here is that most of these methods deal with the communication with database (that requires try-catch blocks) and dealing with the geometric transformations of a glyph. The classes that form Manifold core had very few highly-complex methods, making Manifold core considerable light-weight. Figure 79 shows this characteristic representing the relationship between input and output links and the number of fat methods for main Manifold classes. Core Manifold classes like Viewer and Controller has higher number of links, but do not have any fat methods.

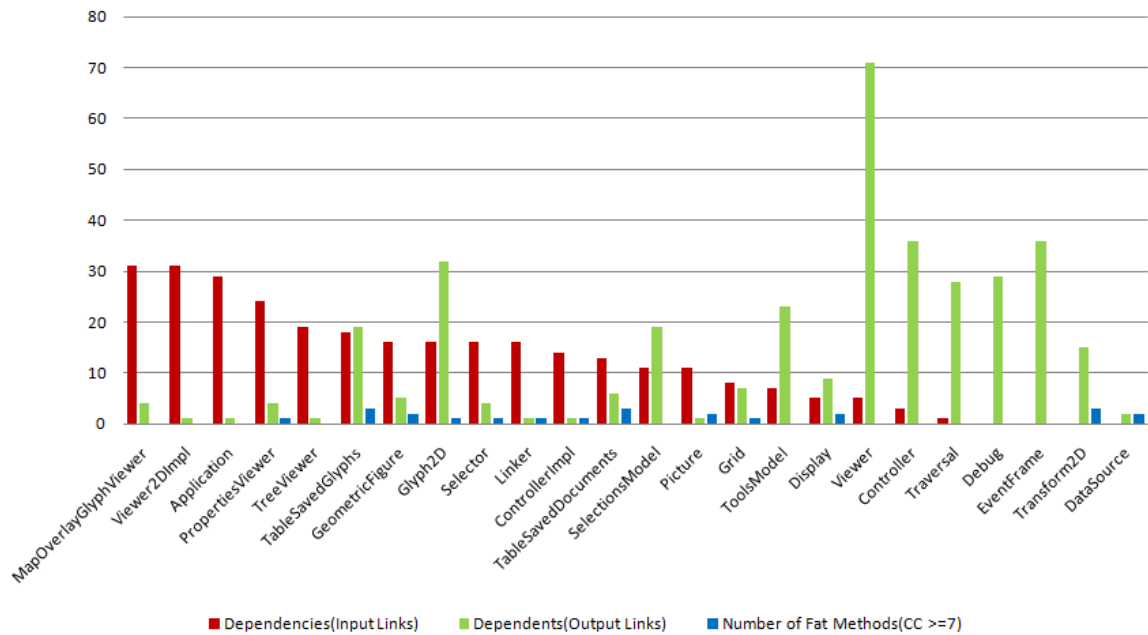


Figure 79: Comparison of Links vs. Fat method in Manifold. Notice that the core classes are not fat.

The average cyclomatic complexity of Manifold was calculated as 1.6204954954954955, making the overall code of the application of very low complexity.

8.2 Performance

This section will explain some of the performance measurement experiments conducted on the newly incorporated features in Manifold as a part of the current thesis. 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. All the performance measurement experiments were conducted on the machine with following configurations:

- **Processor:** Intel (R) Core (TM)2 Duo
- **Speed:** CPU T9300 @ 2.50GHz 2.50 GHz
- **RAM:** 3.5GB
- **Operating System:** Microsoft Windows 7 Professional (32-bit)
- **Java:** Java Development Kit (JDK) 1.6
- **Platform:** NetBeans IDE, Version 6.7.1

8.2.1 Application Loading Time

The loading time of an application determines the amount of time a user has to wait to work with the application. It is determined by the number of features an application provides. More features shouldn't necessary mean more loading time. In this

section we will discuss the comparison between the loading time of the current and the previous version of Manifold.

The main statistics highlighting the differences between the two versions is shown in Table 18.

Table 18: Comparison of current vs. previous version of Manifold

Property	Previous Version	Current Version	Increase (Current - Previous)
Objects (classes, interfaces and packages)	62	127	65
Number of Packages	7	13	6
Number of Relationships i.e. extends, implements, uses (method argument or returned from the method), and contains relationships.	421	809	388
Maximum Dependencies (classes that depends on one class)	31	31	0
Minimum Dependencies (classes that depends on one class)	0	0	0
Average Dependencies (classes that depends on one class)	6.79	6.37	(0.42)
Maximum Dependents (classes on which this class depends on)	42	71	29
Minimum Dependents (classes on which this class depends on)	1	0	(1)
Average Dependents (classes on which this class depends on)	6.79	6.37	(0.42)
Relationship to Object Ratio	6.79	6.37	(0.42)
Number of objects affected by modification of one object	17	16.3	(0.7)
System Stability (calculated as a function of the average number of affected objects)	72%	87%	15%

As accepted, the total number of objects and packages increased as a result of incorporating new features. However, the average number of dependencies remained almost the same, which resulted in an increase in the system stability, as there was an increase in the number of relationships with constant dependency factor. This shows that

Manifold's design is highly scalable. As the number of classes increases, the system stability also increases.

Now, let us see how these two versions compared in terms of their loading time.

This is shown in Figure 80 over a set of five runs.

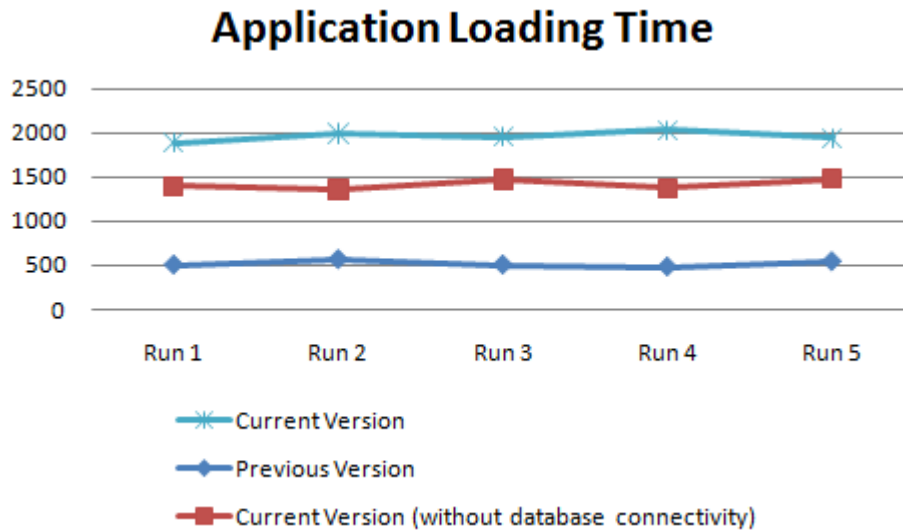


Figure 80: Comparison of loading times of two versions of Manifold

As the figure depicts, there was an increase in the application loading time in the current version. However, I would argue that this increase was not necessarily due to addition of more objects, rather was because of certain new features that required communication with the database (and with web-services, as discussed in the text). These required additional time for parsing the WSDL client specification and creation of Java source files, at the time of loading (using JAX-WS [71]). This increase in loading time was mainly because of multiple-server connections over a network, and would reduce if we run it as a stand-alone application (without using the features which requires this extra connectivity).

Design Issue 8.1: *The loading time of the application can be decreased by running the database and other network connections in separate threads. This would allow the user to start using the workspace while connections are being established.*

8.2.2 Performance: Glyph Saving

Now let us consider the performance of saving a glyph. The method call hierarchy is shown in Table 19. The first step's invocation time is the summation of the sub-steps.

Table 19: Method Call Hierarchy for Save Glyph Menu Item

	Method Name	Invocation Count	Invocation Time
1	manifold.impl2D.menuItems.SaveGlyph.actionPerformed	1	1 s 2.161 ms
1.1	manifold.swing.Viewer2DImpl.paintComponent	1	0.829 ms
1.2	manifold.swing.BaseMenuItem.setViewForRendering	1	0.237 ms
1.3	manifold.swing.Viewer2DImpl.getSelectionsModel	1	0.4 ms
1.4	manifold.SelectionsModel.getSelections	1	0.11 ms
1.5	manifold.swing.Viewer2DImpl.getController	1	0.7 ms
1.6	manifold.EventFrame.<init>	1	0.37 ms
1.7	manifold.ControllerImpl.sendAsyncEvent	1	0.209 ms

This represents a full cycle of saving of a single glyph of type rectangle in the database.

8.2.3 Performance: Saved Glyph Retrieval

Now let us consider the performance of retrieving a saved glyph. The method call hierarchy is shown in Table 20. The first step's invocation time is the summation of the sub-steps.

Table 20: Method Call Hierarchy for Insert Menu Item

	Method Name	Invocation Count	Invocation Time
1	manifold.impl2D.menuItems.ClipArtRectangle.actionPerformed	1	577.460 ms
1.1	manifold.swing.BaseMenuItem.setViewForRendering	1	0.31 ms
1.2	manifold.swing.TableSavedGlyphs.<init>, i.e. constructor		
1.3	manifold.swing.TableSavedGlyphs.getSavedItemFromDatabase	1	475.290 ms
1.4	manifold.swing.TableSavedGlyphs.addTableToPopUp	1	87.493 ms

This represents a full cycle of inserting a single glyph of type rectangle (as saved in previous section) retrieved from the database. The database in these sample runs was located locally on the same machine as the running application, and due to information transfer between two different servers step 1.3 took more time than the others.

8.2.4 Performance: Grouper Tool

Now let us consider the performance of Grouper tool. The method call hierarchy is shown in Table 21.

This represents a full cycle of grouping two glyphs of type rectangle. Some steps like 3.3 takes more time than the other as it represents the step of continuous

manipulation by the user where he/she depresses the mouse button and drags it along the Manifold viewer to draw a selection box on the screen.

Table 21: Method Call Hierarchy for Grouper Tool

	Method Name	Invocation Count	Invocation Time
1	manifold.impl2D.tools.Grouper\$GrouperManipulator.<init>, i.e. constructor	1	0.239 ms
2	manifold.impl2D.tools.Grouper\$GrouperManipulator.grasp	1	0.888 ms
3	manifold.impl2D.tools.Grouper\$GrouperManipulator.manipulate	16	82.469 ms
3.1	manifold.impl2D.InputDeviceEvent.getPoint	16	0.55 ms
3.2	manifold.impl2D.InputDeviceEvent.getScreenPoint	16	0.28 ms
3.3	manifold.impl2D.tools.Selector.drawSelectionBox	32	76.701 ms
3.4	manifold.impl2D.glyphs.Rectangular.clone	1	0.55 ms
3.5	manifold.impl2D.Glyph2D.setProperty	75	0.450 ms
3.6	manifold.impl2D.Glyph2D.setCachedState	1	0.48 ms
3.7	manifold.impl2D.Glyph2D.setSelected	1	0.3 ms
3.8	manifold.impl2D.TransformGroup.addChild	1	0.9 ms
4	manifold.impl2D.tools.Grouper\$GrouperManipulator.effect	1	5.579 ms
4.1	manifold.impl2D.InputDeviceEvent.getPoint	1	0.1 ms
4.2	manifold.impl2D.tools.Selector.gatherSelectionsAndCleanUp	1	2.103 ms
4.3	manifold.impl2D.Glyph2D.setSelected	4	0.6 ms
4.4	manifold.impl2D.GeometricFigure.addChild	2	0.17 ms
4.5	manifold.impl2D.GeometricFigure.getChildren	1	0.10 ms
4.6	manifold.impl2D.tools.Grouper\$GrouperManipulator.setMinimumBoundingShape	1	0.153 ms

8.2.5 Performance: Custom Glyph Insertion

Now let us consider the performance of retrieving a saved glyph. The method call hierarchy is shown in Table 22.

Table 22: Method Call Hierarchy for Custom Glyph Insertion

	Method Name	Invocation Count	Invocation Time
1	<code>manifold.impl2D.menuItems.ClipArtCustomGlyph.actionPerformed</code>	1	3 s 848.477 ms
1.1	<code>manifold.swing.BaseMenuItem.setViewForRendering</code>	1	0.285 ms
1.2	<code>manifold.impl2D.menuItems.ClipArtCustomGlyph.textRecogniserWebService</code>	1	394.671 ms
1.3	<code>manifold.swing.Viewer2DImpl.paintComponent</code>	1	1.133 ms

This represents a full cycle of inserting a custom glyph on Manifold viewer by sending the raw text, “Draw a rectangle of height 20 and width 40 with tx 400 and ty 500 and rotation 45 degree”. Step 1.2 takes more time than the other as it involves information transfer between two different servers (Manifold and text recognition), located on the same machine for our example run.

Chapter 9

Discussion and Future Work

In this thesis, we have discussed the Manifold framework that was developed for creating applications for different domains by providing programming abstractions that are common across domains. Manifold defines four basic abstractions viz. components that encapsulate behavior and appearance of objects, tools supports direct manipulation of components, commands define operations on the components, and external representations define the presentation models for the UI. Manifold is implemented as a library of Java (TM) classes that provides platform independent implementation of the framework and substantially reducing the time and effort for developing various applications. This helps in creating applications with performance and utility as compared with from-scratch counterparts.

We discussed how separate backend applications could be developed and then attached to Manifold frontend using a set of communication protocols. It greatly demonstrated how Manifold can be used to build practical applications by specifying different application semantics on different backend servers and combine them with the frontend Manifold framework (without any modifications to Manifold).

We also discussed how different user interaction techniques could be used on the Manifold framework and how we could extend them for the purpose of Text and Speech recognition.

We discussed the design and implementation issues with the previous versions of Manifold and tried to solve some of them. We also discussed the implementation of new features and enhancing the old ones like tools, glyphs and new property editors. A complete functional Menu bar was developed for Manifold that provided functionalities like inserting, saving, etc. of glyphs on the Manifold workspace.

The Manifold framework provides a natural growth path to the complete fully-functional systems, such as the PowerPoint graphical editor. Another example application that is suitable to build on the Manifold is decision collaboration systems. We implemented this by developing a Map Viewer which could use the pre-existing Manifold features on the top of a map. This can be proved to be very efficient in cross functional planning and execution. It can be used for interactive visualizations and collaborative decision support if a number of user share the same Manifold workspace hosted on a central server, like the commanders and the officers on the battlefield.

Finally, we analyzed Manifold's code and structure in terms of its design and code complexity. We discussed the issues that were inherent in Manifold's design, and looked at the high complexity methods and the reasons for the same. We also calculated the excessive structural complexity [83] of Manifold's code and structure. All the newly added features were compared in terms of their performance for a complete interaction cycle. The results showed that Manifold offers a light weight core that allows easy development and leads to increased performance for its various features. The increase in system stability proved that Manifold has a highly scalable design.

The discussed features were only made possible due to the "generic nature" of Manifold core that allowed us to implement new classes and simply "plug" them into the

core framework without any modification to the framework itself. This is very intuitive in nature and provides an application developer a simplified implementation of various functionality left to his/her imagination. The framework provides reusable functionality in the form of predefined components, commands, and tools. Debugging time was greatly reduced because less code was written to perform major tasks. Our experience is that developing applications on Manifold and providing it with new features is mainly a matter of choosing, designing, and implementing the required components. Hence, significantly less time is spent defining new commands.

In short, the Manifold framework presented here provides a domain-independent implementation of a presentation module. It is meaningful to state that this UI design acts as a *translator* and *interpreter* from the language of human (gestures) to the language of computers. 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.

We have tried to incorporate new features in to Manifold to improve upon its limited functionality from the earlier versions. But what makes the application exciting is that any developer can add as many new features as he/she desires and the type of these features can be left to his/her imagination, such is the design of Manifold. A platform has been created that only has to be enhanced to make it a more suitable application according to one's need. This can be done by developing newer features. While features are one part, the other part would be to enhance the performance of the application in various ways. The next few sections describe the scope of future work that can be done on Manifold to enhance its features and performance.

9.1 Future Work

The new features come into act mainly because of certain design issues that were found during the current work and could not be addressed at that point. Throughout the text we have tried to point out these issues and the first step in the future work would be to address those issues.

9.1.1 New Workspace

In the current implementation we have provided a way to open new workspaces, however, the new workspaces opened, are a part of the main thread, and as soon as the user closes any of the workspace, the main thread gets terminated resulting in the closing of all the opened workspaces (Design Issue 6.1). This could be addressed in the future versions where different workspaces should be part of a separate thread, running differently from the main application itself.

Also, laying out multiple workspaces could be addressed by providing functionalities such as opening them in different tabs, as supported by most of the modern day browsers. One such way to do in Java is using the Tabbed Panes [65]. The application could be put in the tabbed pane, and consequently new workspaces could be opened in new tabs, providing them with mouse and keyboard listeners and with custom components like close buttons on the tabs (that would close only the selected tab that's running in a separate thread).

9.1.2 Keyboard Listeners

Manifold currently doesn't support keyboard events for some of its components. Keyboard events would be very helpful in improving the existing features on Manifold. These could include but are not limited to:

- Selecting multiple glyphs by pressing the Ctrl/Command key (Design Issue 5.2).
This would provide the functionality of grouping arbitrary glyphs, not just the one's selected by drawing the rubber-band object.
- Glyph scaling by using 'Ctrl' + '+' to increase the size and 'Ctrl' + '-' to decrease the size.
- Activating tools/property editors by using keyboard mnemonics (similar to the menu items).

9.1.3 New Features

Although developing new features are left to the creativity and domain usefulness of the application that will use Manifold. But we think that the following features would be good to improve the overall functionality of Manifold.

9.1.3.1 Undo/Redo Features

The undo and redo features allows the user to undo/redo the last performed action. These become even more important when one is using a graphical editor where continuous manipulation is performed on the underlying objects, and the user may want

to revert back or forth on the last performed manipulation. Implementing the undo and redo commands can be done by:

- Remembering the undoable edits
- Implementing the undo/redo commands

This looks very intuitive, however the way it should be implemented on Manifold would require keeping them separate from the application logic. We know that Manifold's Controller supports two way communications from the presentation module to the domain module. It provides support for event frame verbs to support the communication and provides how Manifold communicates with the outside world.

The way I envision this feature to be implemented is through the Controller itself. When the user performs an action commands like add node, delete node, and set properties, these commands are send to the Controller which interprets these actions and fires necessary events to provide the visual feedback to the user. Now when, the user performs an action, if the developer wants that the particular action should support undo/redo commands, then he may create another action verb for the Controller that may look like:

```
public static final String UNDO_ACTION = "undoAction";
```

The application logic may look like:

```
Hashtable slots_ = null;
slots_ = new Hashtable();
slots_.put(EventFrame.VERB, ControllerImpl.UNDO_ACTION);
slots_.put(EventFrame.NODE_ID, nodeId);

//editedSlot is the slot that has been edited
//it will contain the cached state of the glyph
//after performing certain actions
slots_.put(EventFrame.EDITED, editedSlot);
propertiesViewer.getController().sendAsyncEvent(new
EventFrame(slots_));
```


Here, when a user performs certain action on a glyph, an undoable event is also sent to the Controller, meaning that the current action is possible to be reverted back to. Now at the Controller only what is required is to maintain a mapping of the current node id and the associated events that occurred on it. What I mean is that there should be implementation of the following kind:

```
protected UndoManager undo = new UndoManager();
undo.addEdit(e.getEdit());
undoAction.updateUndoState();
redoAction.updateRedoState();
```

Here, when the Controller detects an undoable event, it creates an undo manager class that could contain methods like adding the current slot to the mapping, and updating the current undo/redo states. Keeping a mapping of these states, it would be possible to query the mapping table at any point and get the corresponding glyph state at a particular level of the mapping. One such implementation using the *Command* design pattern is discussed at [66].

9.1.3.2 New Glyph Types

Manifold supports geometric glyph types like Line and Rectangle. We think it would be a nice idea to implement more geometric glyph types such as Arc, Quadric Curve, Cubic Curve, and Arbitrary shapes (these would be very beneficial for the Map Viewer where visualized objects could be arbitrary). These are discussed at [67, 68] and would be relatively easy to implement as all the necessary classes have already been laid out, and these new classes would simply extend to `GeometricFigure` (package

`manifold.impl2D`) and would implement the type related geometric semantics by overriding the `draw()` method.

References

1. HCI: Human Computer Interaction, Online at:
<http://www.hci.iastate.edu/>
2. Wikipedia: User Interface, Online at:
http://en.wikipedia.org/wiki/User_interface
3. Wikipedia: Graphical User Interface, Online at:
http://en.wikipedia.org/wiki/Graphical_user_interface
4. I. Marsic, *Manifold User Interface Framework*, Technical Report, Rutgers University, NJ, 2005.
5. 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.
6. F.Flippo, A.Krebs and I.Marsic, *A framework for rapid development of multimodal interfaces*, Proceedings of the 5th International Conference on Multimodal Interfaces (ICMI 2003), Vancouver, B.C., Canada, pp. 109-116, November 2003.
7. B. B. Bederson, J. Grosjean, and J. Meyer, "Toolkit design for interactive structured graphics," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 535-546, August 2004.
8. Java Camp: Java Design Patterns, Online at:
<http://www.javacamp.org/designPattern/>
9. Microsoft Office Online: PowerPoint, Online at:
<http://office.microsoft.com/en-us/powerpoint/default.aspx>
10. Adobe: Adobe Photoshop, Online at:
<http://www.adobe.com/products/photoshop/family/>
11. R. Eckstein, "Creating Wizard Dialogs with Java Swing," Copyright 1994-2005 Sun Microsystems, Inc., February 10, 2005. Online at:
<http://java.sun.com/developer/technicalArticles/GUI/swing/wizard/index.html>

12. W3Schools, Introduction to XML, Online at:
http://www.w3schools.com/xml/xml_what.asp
13. W3Schools, SOAP Introduction, Online at:
http://www.w3schools.com/soap/soap_intro.asp
14. Adobe: Adobe Illustrator, Online at:
<http://www.adobe.com/products/illustrator/>
15. Sun Java Documentation: Image (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Image.html>
16. Sun Java Documentation: BufferedImage (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/java/awt/image/BufferedImage.html>
17. Sun Java Documentation: Graphics (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Graphics.html>
18. Sun Java Documentation: JOptionPane (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javaw/swing/JOptionPane.html>
19. Sun Java Documentation: JMenuBar (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javaw/swing/JMenuBar.html>
20. Sun Java Documentation: JMenu (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javaw/swing/JMenu.html>
21. Sun Java Documentation: JMenuItem (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javaw/swing/JMenuItem.html>
22. The Java Tutorials: How to Use Menus, Online at:
<http://java.sun.com/docs/books/tutorial/uiswing/components/menu.html>
23. Sun Java Documentation: JFrame (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javaw/swing/JFrame.html>
24. Sun Java Documentation: JTable (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javaw/swing/JTable.html>
25. The Java Tutorials: How to use Tables, Online At:
<http://java.sun.com/docs/books/tutorial/uiswing/components/table.html>
26. Sun Java Documentation: TableModelListener (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javaw/swing/event/TableModelListener.html>

27. Sun Java Documentation: JComponent(Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JComponent.html>
28. Sun Java Documentation: JPanel(Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JPanel.html>
29. Sun Java Documentation: JLabel(Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JLabel.html>
30. Sun Java Documentation: Hashtable(Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/java/util/Hashtable.html>
31. The Java Tutorials: How to write an Action Listener, Online At:
<http://java.sun.com/docs/books/tutorial/uiswing/events/actionlistener.html>
32. The Java Tutorials: How to write a Change Listener, Online At:
<http://java.sun.com/docs/books/tutorial/uiswing/events/changelistener.html>
33. Sun Java Documentation: JTextField(Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JTextField.html>
34. Sun Java Documentation: JButton(Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JButton.html>
35. The Java Tutorials: How to use Combo Boxes, Online At:
<http://java.sun.com/docs/books/tutorial/uiswing/components/combobox.html>
36. The Java Tutorials: How to use Spinners, Online At:
<http://java.sun.com/docs/books/tutorial/uiswing/components/spinner.html>
37. Sun Java Documentation: GraphicsEnvironment(Java 2 Platform SE v1.4.2), Online at:
<http://72.5.124.55/j2se/1.4.2/docs/api/java/awt/GraphicsEnvironment.html>
38. Sun Java Documentation: SpinnerNumberModel(Java 2 Platform SE v1.4.2), Online at:
<http://72.5.124.55/j2se/1.4.2/docs/api/javax/swing/SpinnerNumberModel.html>
39. Sun Java Documentation: NumberFormatException(Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/java/lang/NumberFormatException.html>
40. The Java Tutorials: How to use File Choosers, Online At:
<http://java.sun.com/docs/books/tutorial/uiswing/components/filechooser.html>
41. Wikipedia: Pattern Recognition, Online at:

- http://en.wikipedia.org/wiki/Pattern_recognition
42. Wikipedia: Speech Recognition, Online at:
http://en.wikipedia.org/wiki/Speech_recognition
 43. Java Speech API Programmer's Guide: Speech Recognition, Online at:
<http://java.sun.com/products/java-media/speech/forDevelopers/jsapi-guide/Recognition.html>
 44. MSDN Documentation: Noise Words, Online at:
[http://msdn.microsoft.com/en-us/library/ms693206\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms693206(VS.85).aspx)
 45. H.C. Andrews, Introduction to Mathematical Techniques in Pattern Recognition, Wiley-Interscience, a Division of John Wiley & Sons Inc., New York, 1972.
 46. S. Theodoridis, K. Koutroumbas, Pattern Recognition, Third Edition, Elsevier (USA), 2006
 47. N.A. Campbell, "Shrunk Estimators in Discriminant and Canonical Variate Analysis", Applied Statistics, Vol. 29, No. 1, pp. 5-14, 1980
 48. R.P.W. Duda and E. Backer, "Discriminant analysis in a non-probabilistic context based on fuzzy labels", in Pattern Recognition and Artificial Intelligence, Edited by Gelsema, E.S. and Kanal, L.N., Elsevier Science Publishers B.V., pp 229-235, 1988.
 49. T. Joachims, "Making large-scale SVM learning practical", in Scholkopf, B., Burges, C.J.C. and Smola, A.J., editors, *Advances in Kernel Methods – Support Vector Learning*, MIT Press, Cambridge, USA, 1998.
 50. JSON (Java Script Object Notation): JSON in Java, Online at:
<http://www.json.org/java/>
 51. Sun Java Documentation: XMLDecoder (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/java/beans/XMLDecoder.html>
 52. Java.net: Building Maps into Your Swing Application with the JXMapView, Online at:
<http://today.java.net/pub/a/today/2007/10/30/building-maps-into-swing-app-with-jxmapviewer.html>.
 53. The Java Tutorials: How to Use Layered Panes, Online at:
<http://java.sun.com/docs/books/tutorial/uiswing/components/layeredpane.html>
 54. Sun Java Documentation: JToggleButton (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JToggleButton.html>

55. Swing Labs: Java Desktop Technology, Online at:
<http://swinglabs.org/>
56. Wikipedia: Glyph, Online at:
<http://en.wikipedia.org/wiki/Glyph>
57. E. Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, Inc, Reading, MA, 1995.
58. 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.
59. S. Churchill, *Structured graphics in Fresco*, C++ Report, pp. 61-68. 3, March/April 1995.
60. Java.net: Mapping Mashups with the JXMapView, Online at:
<http://today.java.net/pub/a/today/2007/11/13/mapping-mashups-with-jxmapviewer.html>
61. G. Krasner and S. Pope, "A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, vol.1, no.3, pp.26-49, August/September 1988.
62. 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, 2005.
63. Sun Developer Network (SDN): Developer Resources for Java Technology, Online at:
<http://java.sun.com/>
64. MIT-AI Laboratory Memo 306, June 1974: FRAMES, A Framework for Representing Knowledge, Online at:
<http://web.media.mit.edu/~minsky/papers/Frames/frames.html>
65. The Java Tutorials: How to Use Tabbed Panes, Online at:
<http://java.sun.com/docs/books/tutorial/uiswing/components/tabbedpane.html>
66. JAVAWORLD: Add an undo.redo function to your Java apps with Swing, Online at:
<http://www.javaworld.com/javaworld/jw-06-1998/jw-06-undoredo.html>
67. The Java Tutorials: Drawing Geometric Primitives, Online at:
<http://java.sun.com/docs/books/tutorial/2d/geometry/primitives.html>

68. The Java Tutorials: Drawing Arbitrary Shapes, Online at:
<http://java.sun.com/docs/books/tutorial/2d/geometry/arbitrary.html>
69. Wikipedia: Cyclomatic Complexity, Online at:
http://en.wikipedia.org/wiki/Cyclomatic_complexity
70. Sangwan R.S., Vercellone-Smith P, Laplante P.A., “Structural Epochs in the Complexity of Software over Time”, *Software, IEEE*, vol.25, issue.4, pp.66-73, 2008.
71. Java.net, JAX-WS Reference Implementation, Online at:
<https://jax-ws.dev.java.net/>
72. Sun Java Documentation: AffineTransform (Java 2 Platform SE v1.4.2), Online at:
<http://java.sun.com/j2se/1.4.2/docs/api/java/awt/geom/AffineTransform.html>
73. B. Shneiderman, “The limits of speech recognition,” *Communications of the ACM*, vol. 43, no.9, pp. 63-65, September 2000.
74. B. Shneiderman, *Leonardo’s Laptop: Human Needs and the New Computing Technologies*, The MIT Press, Cambridge, MA, 2002.
75. Sphinx-4: A speech recogniser written entirely in the Java(TM) programming language, Online at:
<http://cmusphinx.sourceforge.net/sphinx4/>
76. A. Shaikh, S. Juth, A. Medl, I. Marsic, C. Kulikowski, and J. Flanagan, “An architecture for multimodal information fusion,” *Proceedings of the ACM Workshop on Perceptual User Interfaces (PUI '97)*, pp. 91-93, Banff, Alberta, Canada, October 1997.
77. Sun Java Documentation: JDesktopPane (Java 2 Platform SE v1.4.2), Online at:
<http://download.oracle.com/javase/1.4.2/docs/api/javax/swing/JDesktopPane.html>
78. Sidhanti, R. 2009. Enhancements of the generic Manifold user interface, New Jersey. Masters Thesis, Rutgers University, New Brunswick. 43 p.
79. Wikipedia: Parsing, Online at:
<http://en.wikipedia.org/wiki/Parsing>
80. Python Programming Language – Official Website, Online at:
<http://www.python.org/>
81. alphaWorks: Structural Analysis for Java, Online at:
<http://www.alphaworks.ibm.com/tech/sa4j>

82. CyVis: Software Complexity Visualizer, Online at:
<http://cyvis.sourceforge.net/>
83. Headway Software: XS – A measure of Structural Over-Complexity
<http://www.headwaysoftware.com/products/structure101/XS-MeasurementFramework.pdf>
84. Spike Developer Zone: Measuring Code Complexity Metrics, Online at:
http://developer.spikesource.com/wiki/index.php/Measuring_Code_Complexity_Metrics
85. CyVis: What is Cyclomatic Complexity, Online at:
http://cyvis.sourceforge.net/cyclomatic_complexity.html
86. S. Valverde, R. Ferrer Cancho, and R. V. Solé, “Scale-free networks from optimal design,” *Europhysics Letters*, vol. 60, no. 4, pp. 512-517, November 2002. Online at:
<http://www.santafe.edu/media/workingpapers/02-04-019.pdf>
87. Wikipedia: JSON-RPC, Online at:
<http://en.wikipedia.org/wiki/JSON-RPC>
88. SourceForge: Girders, Online at:
<http://sourceforge.net/projects/girders/>
89. SourceForge: Strandz, Online at:
<http://sourceforge.net/projects/strandz/>