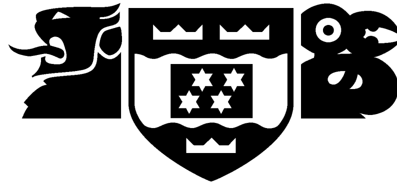
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematics, Statistics and Computer Science
Te Kura Tatau

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

J-grams

Ramon Steenson

Supervisor: Peter Andreae

Submitted in partial fulfilment of the requirements for
Bachelor of Information Technology.

Abstract

This project looked at the construction of a diagram editor called J-grams with particular emphasis on the different phases of the development lifecycle. More specifically the analysis phase which surveyed existing diagram editors to gather requirements, the design phase which discussed the design decisions chosen and the implemention phase, describing some of the implementation details. This report concludes with recommendations for future work in regards to this project.

Acknowledgments

I would like to thank my supervisor Peter Andreae for his assistance, technical expertise and his ability to reduce the scope of this project when needed.

Contents

1 Introduction **1**
1.1 Overview of the Project . . . 2
1.2 Goals of the Project . . . 2
1.3 Approach to the Project . . . 2
1.4 Structure of the Report . . . 2
1.5 Definition of Terms . . . 3
1.5.1 Defining What a Diagram is . . . 3
1.5.2 Using Vectors to Represent a Diagram . . . 3
1.5.3 Storing a Diagram . . . 3

2 Analysis **5**
2.1 Survey of Available Diagram Editors . . . 5
2.1.1 idraw . . . 5
2.1.2 Microsoft Visio 2003 . . . 6
2.1.3 Xfig/Jfig . . . 6
2.1.4 Dia . . . 7
2.1.5 yEd . . . 7
2.1.6 uDraw . . . 7
2.1.7 Overview of Diagram Editor Survey . . . 8
2.2 File Formats Survey . . . 8
2.2.1 SWF . . . 9
2.2.2 WMF . . . 9
2.2.3 PS, EPS and PDF . . . 9
2.2.4 SVG . . . 9
2.2.5 FIG . . . 10
2.2.6 Export Functionality . . . 11
2.2.7 Overview of File Formats . . . 11

3 Requirements **13**
3.1 Primary Requirements . . . 13
3.2 Secondary Requirements . . . 14
3.2.1 Shape Queue . . . 14
3.3 Requirements Met . . . 15

4 Design **17**
4.1 User Interface Prototype . . . 17
4.2 Class Diagram . . . 20
4.2.1 Pen . . . 20
4.2.2 Shapes . . . 21
4.3 Design Patterns . . . 21

4.3.1 Model-View-Controller Pattern 21
4.3.2 Observer Pattern 22
4.4 SWT Library 22
4.5 SVG Integration 22
4.5.1 svgsalamander 22
4.5.2 Batik 23

5 Implementation **25**
5.1 Shape Intersections 25
5.2 Quadtree 26
5.2.1 Structure 26
5.2.2 Performance 26
5.2.3 Memory Optimization 26
5.2.4 Zooming 26
5.3 Unbounded Quadtree 27
5.3.1 Structure 27
5.3.2 Example 28
5.4 Backing Store 28
5.4.1 Double-Buffering 28
5.4.2 Triple-Buffering 28
5.4.3 Performance Still Slow 29
5.5 Matrix Transformations 29
5.5.1 The Bounding Box of a Rotated Ellipse 29
5.5.2 Implicit Differentiation 29
5.5.3 Matrix Transformations and a Rotated Ellipse 30
5.6 SVG Support 30
5.6.1 Implemented SVG Parser Functionality 30
5.6.2 Extending the SVG Specification 31
5.7 Creating Custom Widgets for SWT 31
5.7.1 Extending SWT 31
5.7.2 Displaying and Hiding the Colour Picker 32

6 Conclusions **33**
6.1 Future Work 33
6.1.1 R-Tree 33
6.1.2 Shape Queue 33
6.1.3 Animation Support 33

Chapter 1

Introduction

J-grams is a diagram editor that is programmed in Java and uses the Standard Widget Toolkit (SWT) to provide its user interface and the canvas to draw the shapes on. Currently J-grams runs on Windows and UNIX based systems and can draw a wide range of diagrams, with its core focus being able to draw simple diagrams such as tree structures. Functionality will be included later which adds specific diagram shapes such as UML as the focus was to create a solid foundation which could be extended. Diagrams and normal SVG files can be saved, loaded, edited and can also be exported to a wide range of image formats.

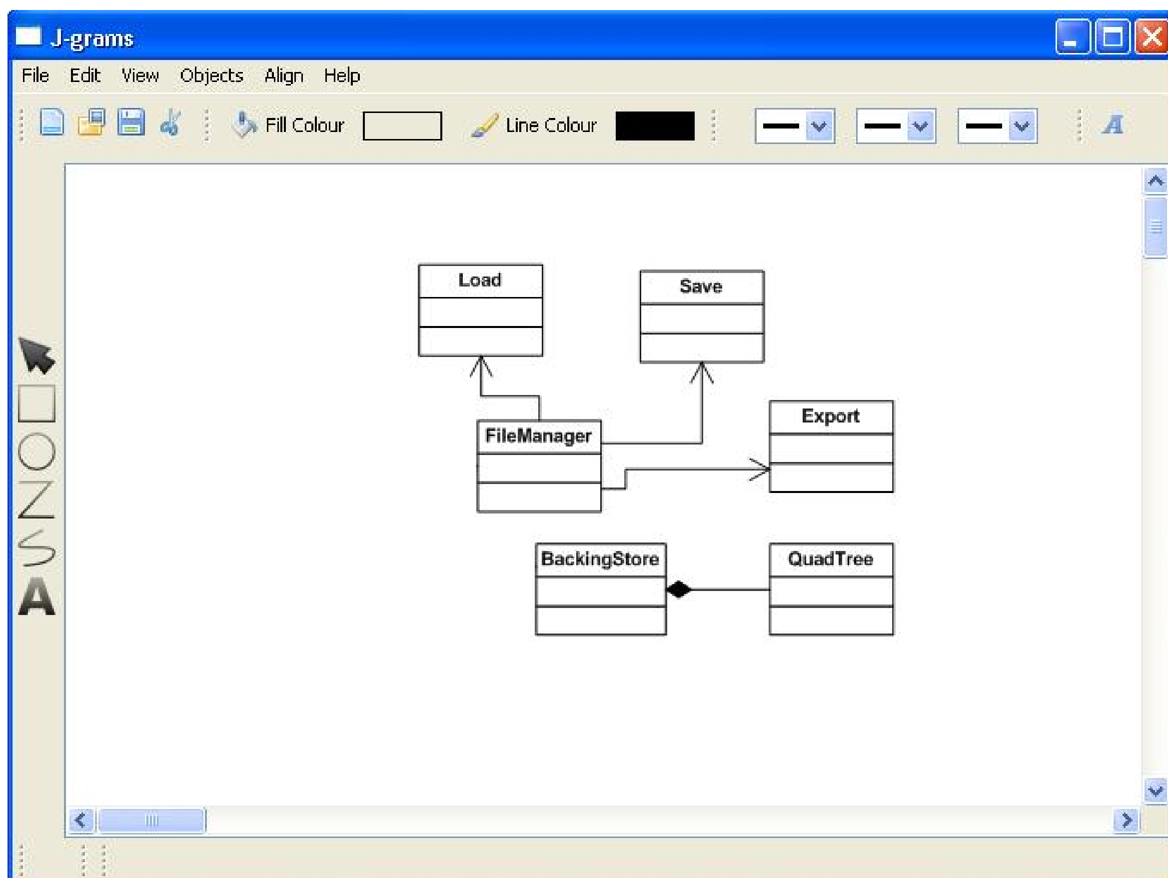


Figure 1.1: User interface of J-grams

1.1 Overview of the Project

This project is based on the construction of a diagram editor for the use of all computer users. The diagram editor focuses on the construction of generalised diagrams such as tree structures. However by implementing a good set of features and an easy to use user interface, it can be extended upon in the future to support specific types of diagrams. These features come from a survey into the existing diagram editors. These diagram editors have good features, but none of them have all the desirable good features. Some of them also have features that are not necessary for the target users. This project finds the good features as well as creating new ones and combines them into a functional diagram editor called J-grams.

1.2 Goals of the Project

The objectives of this project are:

- Survey existing diagram editors and file formats to generate a list of requirements
- Conduct a simple usability analysis for a prototype user interface
- Design a diagram editor using UML
- Practise good project management skills
- Implement the requirements gathered

1.3 Approach to the Project

The first objective was to survey the existing diagram editors and file formats. This is done by surveying the existing diagram editors and file formats to find what features they contain and whether they are good. A requirements document is then created which defines what features J-grams should contain and also what file format to store the diagram in.

The usability anaylsis objective is completed by doing a user analysis with potential users and creating a usability test with a prototype user interface. UML is then used to design J-grams based on the requirements gathered, then using the agile development methodology J-grams is implemented using good project management.

1.4 Structure of the Report

This report steps through the software development cycle that was used in this project, specifically focusing on the analysis, design and implementation phases.

The first chapter describes the anaylsis phase which contains the survey into already existing diagram editors and the file formats which are capable of storing a diagram. The design phase constructs J-grams using UML and describes some of the design problems and solutions that occured during this stage. The final chapter describes the implementation phase looking at some of the implementation problems that needed to be addressed and discusses the solutions that were implemented.

1.5 Definition of Terms

1.5.1 Defining What a Diagram is

A diagram is typically composed of several basic shapes which are then transformed and connected with other shapes. These basic shapes are:

- Line
- Curve
- Rectangle
- Ellipse
- Polygon
- Poly-line

These basic shapes are used in a particular way to represent the actual information a diagram encapsulates. For example a UML class diagram represents a class with a larger rectangle composed of three smaller rectangles inside of it. These rectangles then contain the information the diagram represents.

1.5.2 Using Vectors to Represent a Diagram

A diagram is a vector based image which means that it is represented by mathematical equations. In comparison to raster based images which use groups of pixels to represent images, the clarity and file size are far superior when vector graphics are used. In vector graphics the mathematical equations can be thought of as a set of geometrical primitives; namely points, lines, curves, and polygons. These primitives used in a diagram can represent all of the basic shapes that are needed to create a diagram.

1.5.3 Storing a Diagram

Storing a diagram requires a lossless file structure which encompasses these primitives in such a way that that can be easily interpreted to reconstruct the actual diagram those primitives represent. Traditional file structures used a raster based structure that stored the actual pixel (RGB) data. A vector based image has the advantage of storing only the primitive commands to draw the shapes. This leads to smaller sizes and easier integration with other programs.

Chapter 2

Analysis

The analysis phase for this project consists of surveying the current diagram editors and file formats to construct a requirements list. It is important that a wide range of programs and formats be surveyed as the features that are required for the construction of J-grams will be easier and more complete. This stage is an important step for any software project as it assesses the feasibility of the project without commiting too much resources.

2.1 Survey of Available Diagram Editors

There are already a number of existing diagram editors that are available for users to use. Some of these diagram editors were surveyed to gather which requirements are needed and what a diagram editor should do. The survey looked at the following features in each diagram editor

- The user interface
- How shapes are drawn
- How connectors are used
- What kinds of styles can be applied to shapes
- The layout algorithms available
- How easy it is to make a modification to a shape

These areas will provide a list of requirements which will then be used for the actual implementation of J-grams.

2.1.1 idraw

idraw[5] is a simple to use diagram editor that is freely available for anybody to use. It is based on a toolkit which is dated when compared to other editors. It focuses more however on drawing diagrams. Its user interface was easy to grasp for a first time user and drawing shapes was an easy "point and click" situation. Its user interface uses the original X11 libraries to draw itself, thus making drawing shapes very fast but leading to an unattractive interface.

Drawing connectors between shapes is difficult with idraw because it does not support the notion of actually connecting two shapes with a line. A line can be simply drawn between shapes but when one of those shapes moves the connection is lost between the two. Idraw also suffers from not being able to add arrow heads to the end of lines easily.

Once shapes are drawn it is difficult to change their properties as it does not provide helpful visual assistance. It is also not clear whether or not you are actually clicking on the shape unless you are directly on top of the line which makes selecting a single shape difficult.

2.1.2 Microsoft Visio 2003

Microsoft Visio 2003[6] is a commercial diagram editor that is sold as part of Microsoft Office. It is a generalized diagram editor that has the capabilities to draw a large number of different diagrams. The editor presents a list of available diagram types for a user to choose which then allows them to try and find which diagram they want to draw.

When a diagram has been selected it changes the available shapes to draw only those shapes which can be used in that diagram. It is extremely difficult to change another shape outside of what the current diagram view provides and requires a user enter search terms rather than presenting a list of available shapes. This has problems when drawing simple diagrams as it is difficult to add a rectangle or an ellipse. This means that more complex diagrams cannot be constructed from the basic shapes. It is a good program for beginners to use, but for the more intermediate and experienced users it does not support the freestyle drawing that is required by those users. When drawing a shape there has to be a drag and drop movement so that only one shape can be drawn at once. This is a drawback because of the lost time moving back and forwards selecting the same shape again to redrawn another one.

Modifying an already drawn shape is simple, and there is good visual feedback to show the user which shape has been selected. There are options to move and rotate selected shapes. The connector tool is very powerful in Visio as it allows for easy connector creating and editing. It is possible to connect shapes via a central point and any number of pre-defined points around a shape. Once connected they stay connected even when the shape is moved. Connectors can be easily moved and routed around other shapes to reach its intended connection.

2.1.3 Xfig/Jfig

Xfig[8] is very similar in most respects to idraw but it is more powerful in the amount of features it supports. Xfig uses the X11 libraries so is only supported on UNIX systems and a port of Xfig called Jfig[2] uses the libraries in Java so that it is available on all systems the Java virtual machine runs on.

The interface that xFig uses is modal and it is difficult to know which button on the mouse performs which functions. Moreover because of the vast amount of features presented to the user they can get easily confused. For a first time user, this program seems very difficult to use, but it is very powerful for the more experienced users. It supports LaTeX-formatted text objects which make it ideal for academics to use.

This program focuses on the basics of drawing diagram at the lowest level possible and does not provide the notion of connectors which makes drawing diagrams difficult. The user interface does not provide good visual feedback on what the cursor is actually doing and what is selected. Also it is sometimes difficult to determine what function will be performed when clicking on icons to do specific tasks. The icons have to go as far as telling the user in a label what function they provide rather than visual metaphors.

2.1.4 Dia

Dia[1] is an open source diagram editor that can draw a large amount of diagrams. Its user interface is simple to use and diagrams can be easily constructed.

It supports many diagram types through its plug-in interface which allows additional shapes to be added to the program. For example it can draw UML, ERD, and electrical circuit diagram. It includes this functionality in an interesting way as it does not interfere with the usability and core functionality of the program. These extra shapes can be selected from a drop down box which shows the available categories of shapes. While the interface still provides the ability to draw diagrams using the basic shapes. Diagram specific shapes are easily changed by switching the category in the drop down box.

Dia provides good visual feedback and has good connector support which allows complex curves and poly-lines to be created so that connectors can be routed around other shapes. When a shape is selected it is easy to modify the dimensions and styles and each shape provides a properties box for fine control of shape properties for expert users.

2.1.5 yEd

yEd[9] is primarily used for drawing graphs and supports automatic layout algorithms using graph theory and routing. It is Java based so is platform independent, however performance was slower on UNIX based machines when compared to Windows. It also supports the construction of UML diagrams. However this support is limited and can be difficult to use.

The program uses a commercial SWT widget set and adopts the native look of the platform it is run on. Flickering occurs a lot when using the program when the window has to redraw itself which leads to a bad user experience. The canvas that is used also flickers when a graph or diagram is being drawn.

yEd is set out in a functional way as it keeps the user in the same context all the time when drawing. Shapes are able to be styled by simply changing the options in the options side bar on the right side of the screen. Drawing shapes and selecting them was difficult for first time users and it was also not clear how to change the default shape from a square box to another shape.

yEd specialises in the construction of graphs and has layout features that support this. Connectors are easily joined and stuck to a shape, and shapes also snap to a grid when placed onto the canvas. yEd offers advanced features for automatically aligning a graph to certain algorithms such as being hierarchical, as a tree, circular or orthogonal. These algorithms use graph theory. Sometimes these features did not work very well and with large graphs they slowed the computer significantly so that yEd became unusable.

2.1.6 uDraw

uDraw[7] focuses on drawing graphs and not specifically diagrams. Its graph drawing functionality is comparable to those features of yEd which include automatic layout handling using graph theory.

It has layout algorithms to improve the readability of graphs which is similar to yEd. The algorithms appear to work correctly on the test graphs that were drawn. However it only focuses on simple layouts unlike yEd which contains a lot of specialised layout algorithms.

It uses a vertical grid of levels to draw diagrams which makes drawing graphs easy. Nodes can be moved from level to level, and edges connected to it will follow. This layout technique is simple and is used to support its layout algorithms.

2.1.7 Overview of Diagram Editor Survey

From the survey of the diagram editors there are several features which are needed to help draw a diagram. These features are:

- Support for the basic shapes such as rectangle, ellipse, poly-lines, and curves
- Connector tools so that a shape can be connected to another
- Ability to add arrow heads to the ends of the connections
- Good visual feedback when modifying and selecting shapes
- Provide an uncluttered user interface which supports beginner users through to expert users
- Allow shapes to be grouped and aligned using different algorithms

These features provide a set of core requirements that a diagram editor should contain. The primary requirements are the initial features the diagram editor will contain at the end of development. The secondary requirements are experimental features that will be implemented if there is enough time at the end of the project. It is regarded that the primary requirements are the top priority in J-grams.

2.2 File Formats Survey

A required feature for all diagram editors is the ability to save a diagram to a file and to load a diagram from a file. This requires a file format which allows the structure of the diagram to be captured, specifically a lossless format which is focused around vector graphics. The following is a list of file formats that support vector based graphics:

- Small Web Format (SWF)
- Windows metafile (WMF)
- Postscript (PS)
- Encapsulated Postscript (EPS)
- Portable Document Format (PDF)
- Scalable Vector Graphics (SVG)
- fig

Some of these file formats are not capable of reconstructing a complete diagram as they do not preserve the connections between shapes. Each format was investigated look to find out which is the best file format to store a diagram in, focusing particularly on the file formats specification, how much is it supported in the real world, and how easy is it to integrate into a diagram editor.

2.2.1 SWF

Small Web Format is a vector based image format that was developed by Adobe Flash. It is a proprietary file format that supports animation and scripting which makes it good for web based solutions. SWF files are compressed and unreadable as a final product and consequently there are not many programs that support the drawing and editing of SWF files. Although this is an open format which has its specification freely available, a license is required to play SWF files. This presents a problem as an initial investment is required to load SWF files into a diagram. Also a significant amount of time would be needed in creating a library which outputs correct SWF files.

In terms of functionality SWF files support all the drawing primitives that are needed to contain a diagram. However this specification is mainly suited for web based solutions and as such there are a lot of features which are not necessary for a diagram editor. In fact an entire diagram editor could be constructed using SWF.

2.2.2 WMF

Windows metafile is in old standard which was created by Microsoft for use in their Windows operating system. It supports vector graphics and the inclusion of raster graphics which makes it able to contain a diagram. However it is a closed specification and does not support application dependent mark-up which diagrams need to contain the actual connections between shapes. A WMF file at its lowest form is a list of drawing primitives such as move-to, line-to and it is possible to convert a WMF file to an SVG file because of this similarity.

2.2.3 PS, EPS and PDF

These three file formats have the functionality to contain diagrams but they are mainly suited towards the layout of text. They are also very complex and as such are not suitable to contain diagrams.

2.2.4 SVG

SVG is an XML based file format specification that describes two-dimensional vector based images. SVG was created by the World Wide Web Consortium. SVG is currently supported by most commercial vector based packages and is currently implemented into Mozilla Firefox, Opera, and Internet Explorer (via a plug-in). This makes it an ideal file format because it is widely supported and there are many viewers that can render SVG files.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
xmlns="http://www.w3.org/2000/svg" version="1.1">
   <rect x="1" y="1" width="1198" height="398"
fill="none" stroke="blue" stroke-width="2"/>
<rect x="400" y="100" width="400" height="200"
fill="yellow" stroke="navy" stroke-width="10"  />
</svg>
```

Figure 2.1: SVG code example

The SVG specification requires implementations of it to conform to the following file format standards:

1. SVG Document Fragments
2. SVG Stand-Alone Files
3. SVG Included Document Fragments

Additionally a program that manipulates a SVG file must conform to one of the following standards, of which depends on how it manipulates an SVG file.

1. SVG Generators
2. SVG Interpreters
3. SVG Viewers

An SVG Generator requires that it create a document that conforms to one or more of 1, 2, or 3. While an SVG Interpreter must parse an SVG document correctly it is not required to understand the semantics of all the features correctly. This is particularly important as it allows an implementation to not implement all of the features described by the specification, but it must be able to parse the document. Additionally an SVG Viewer describes how an SVG document must be rendered on to a medium.

J-grams would have to conform to all three file format standards, and all three manipulation standards. Implementing those areas would allow J-grams to successfully load, save, view, and manipulate a SVG document. There are some important requirements that it must adhere to and they are mostly stated in the SVG Viewer area. It must have the ability to zoom and pan, support PNG and JPG image formats, must have alpha channel blending, and must support base64-encoded content encapsulated in the SVG document.

SVG files have the capability to contain application dependent mark-up which can be used to store information such as the connections between shapes. This is called a foreign namespace in the SVG specification. When other viewers try and render the diagram they will simply ignore this mark-up if they conform to the standard of being classified as a viewer.

2.2.5 FIG

Fig is the file format which xfig and jfig use to represent diagrams. It is a simple text based format that uses words to describe an object and integers and floating point numbers to describe its size, position, and style. The file format supports the basic shapes and provides a compound shape which adds the capability to group shapes together so that a shape can contain any number of shapes.

There is no native support for application dependent structure, which would be necessary to connect two shapes together with a connector. To change fig would require extending the already established specification while breaking compatibility with other applications. Another drawback is that the styles available are very restrictive and only a certain range of fonts and colours can be used. However fig does support the inclusion of LaTeX.

Despite these implications it is a very simple and elegant file format for describing diagrams, but the limits on the capabilities of the format place limitations on the features that could be implemented in this project.

2.2.6 Export Functionality

ImageMagick is a library which can convert an image from one format to another, for example it can convert an SVG file to an EPS file for easy inclusion into LaTeX documents. Including this functionality is important as it doesn't constrain users to use purely SVG files.

2.2.7 Overview of File Formats

There are many file formats which are capable of drawing a diagram when viewed. However the ability to extend them to support application dependent code was a feature that most did not support. In the end SVG was chosen because of the ability to extend so that the diagram will still display in other SVG viewers.

Chapter 3

Requirements

The requirements are gathered from the analysis stage, in particular from the survey of existing diagram editors and file formats. The requirements are a formal document which states what requirements the project must complete. J-grams has two sets of requirements, the first being the primary requirements which are the core features that are to be implemented first and be completed by the end of the project. The second requirements are the secondary requirements which are features which are to be implemented if there is enough time. These secondary requirements also define future work that will extend the functionality of J-grams.

3.1 Primary Requirements

The primary requirements are summarised below:

1. Support the drawing of the following shapes:

   - Rectangle, ellipse, poly-lines, Bezier splines, polygons

   Aside from drawing a shape, shapes can be resized, moved, rotated and can have a label associated to it. The label will be initially positioned at the centre of the shape which is defined as the rectangular area it occupies.

2. Shapes can be styled with the following options:

   - Line and fill colour
   - Line style

3. Each shape has the ability to be grouped with other shapes to create a composite shape. A change on the group will be reflected on each shape contained in the group.

4. A shape can be connected to another shape. This connector can be a poly-line, Bezier spline or a single line. A connector is an extension of a basic shape and contains information to find the intersections with its connecting shapes. By finding the intersections an arrow head can be applied to each end of the shape. A wide variety of pre-programmed arrow heads can be chosen from basic arrow heads to diagram specific heads such as those required in UML diagrams.

5. A shape has a pre-defined number of connector points that connectors can connect up to. Once a connector is connected to one of these points a shape can be moved, resized, rotated and the connector will stay connected. These points are defined on the edges

of the shapes. Also a special point is defined in the centre of a shape which acts as a dynamic connection point. It is dynamic in that the connector is connected to that point, but the intersection on the outside of the shape is found, which forms a line to the central point.

6. When a shape is selected, visual aids are drawn that allow the shape to be resized, rotated, and moved. When a shape is resized, rotated, or moved, an outline of the change is drawn as the user makes changes. A group of selected shapes can be aligned based on their rectangle areas they occupy. These include:

   - Left and right sides
   - Tops of bottoms
   - Vertical and horizontal centres
   - Abut left, right, up and down

   These allow shapes to be quickly aligned instead of moving them individually.

7. A diagram can be saved and loaded to and from an SVG file. A normal SVG file can be read and modified. A diagram can also be exported to numerous image formats using the ImageMagick libraries to convert a SVG file.

8. The user interface supports all of the requirements defined above. It supports beginner users through to expert users. Expert users require keyboard shortcuts, while beginner users need an easy to use user interface with icons that use visual metaphors.

3.2 Secondary Requirements

The secondary requirements are summarised below:

1. Labels can be moved to different positions other than the central point of the shape. They can be attached to points on a shape such as a corner so that when a shape is moved, resized or rotated, the label area is changed as well. Labels can also follow a path for example a label can follow a curve.

2. Allow the addition of LaTeX to be inserted into a label

3. A grid is displayed on the canvas which provides assistance to align objects, and allows accurate dimensions for shapes to be chosen. This grid provides visual assistance when moving, rotating, and resizing objects so that the shape can be aligned to be the same size, or rotation as another shape. The grid also provides the ability for shapes to be snapped to certain key positions when the shape is moved, rotated, or resized. This snapping feature can either be permanently disabled or temporarily disabled by holding down the ctrl key.

3.2.1 Shape Queue

The shape queue is an experimental feature to help expert users draw diagrams faster. It is a priority queue where shapes are added to when they are drawn or grouped together to form a composite shape. An expert user can then select the recently drawn shapes via a display that is shown on demand.

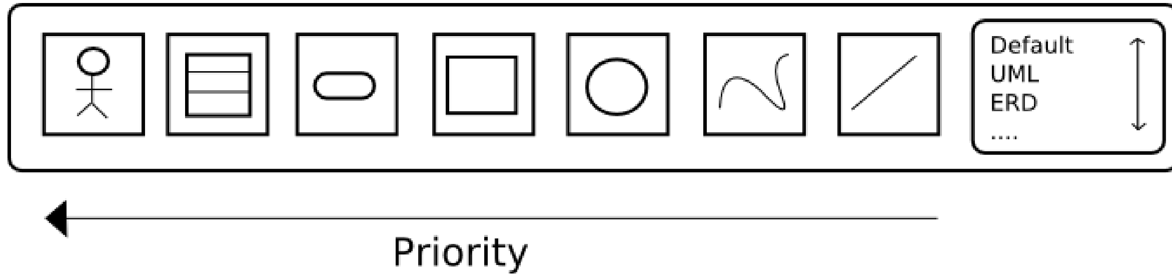


Figure 3.1: Shape Queue prototype design

When a shape is used from the queue, its priority count is increased so that it moves up the queue towards the top to the more recently used shapes. This would benefit expert users that wish to create a lot of shapes based on a certain style for example a tree diagram.

The queues state is saved when J-grams is exited, and the state is reloaded when the program is loaded. In addition to this, diagram profiles could be added with pre-defined shapes which are shapes for specific diagram types such as UML diagrams. Specific diagram types could then be supported without conflicting with the core functionality of the diagram editor.

3.3 Requirements Met

This table summarises the major features discussed above and whether or not they were implemented:

Primary Requirements	**Implemented?**
1. Shapes	Yes
2. Styles	Yes
3. Grouping	Yes
4. Connectors	Yes
5. Connector Points	Yes
6a. Visual Aids	Yes
6b. Alignment Algorithms	Yes
6c. Snap to Grid	Yes
7. SVG Saving/Loading	Yes
8. User Interface	Yes
Secondary Requirements	
1. Labels Positioning	No
2. LaTeX support	No
3. Grid support	60%
4. Shape Queue	No

Table 3.1: Summarised table of features implemented

Chapter 4

Design

The design phase creates an abstract model of a diagram editor based on the requirements gathered previously. This phase is important for any project as it allows a project to be modelled and analyzed for problems that may occur during the implementation phase which can be more costly and harder to fix.

4.1 User Interface Prototype

A user analysis was performed which analysed some of the potential users of J-grams. This analysis found out how users would interact with J-grams, what diagrams they would like to produce, what user interfaces they liked and how they would use it together with other tools. Having such an analysis provides a direction for the design of J-grams, as it will help to design the user interface and functionality to a target audience.

The test users used in this analysis were from a wide background, including a mechanical engineer, a human resource manager, an IT manager and a software developer. The users mostly wanted to draw diagrams that were related to their field of work, focusing on the rapid drawing of rough diagrams. For example the mechanical engineer was interesting in a diagram editor that would allow rough informal drawing of plans for a mechanical system such as the construction of ducting systems. Users also wanted a simple interface which was easy to use and not cluttered with unnecessary features. When shown some of the existing diagram editors most found that idraw was the most easy to use. Integration with other programs was mixed, the business type users wanted integration with the Microsoft Office suite of programs, while the technical users wanted lots of export functionality and printing facilities.

With the feedback from the users a interface prototype was created. A user interface prototype helps to visualise the final product, but more importantly tries to find difficulties that the different kind of users may face when using J-grams. The focus was on creating a simple interface that captures the functional requirements that were defined in the requirements analysis.

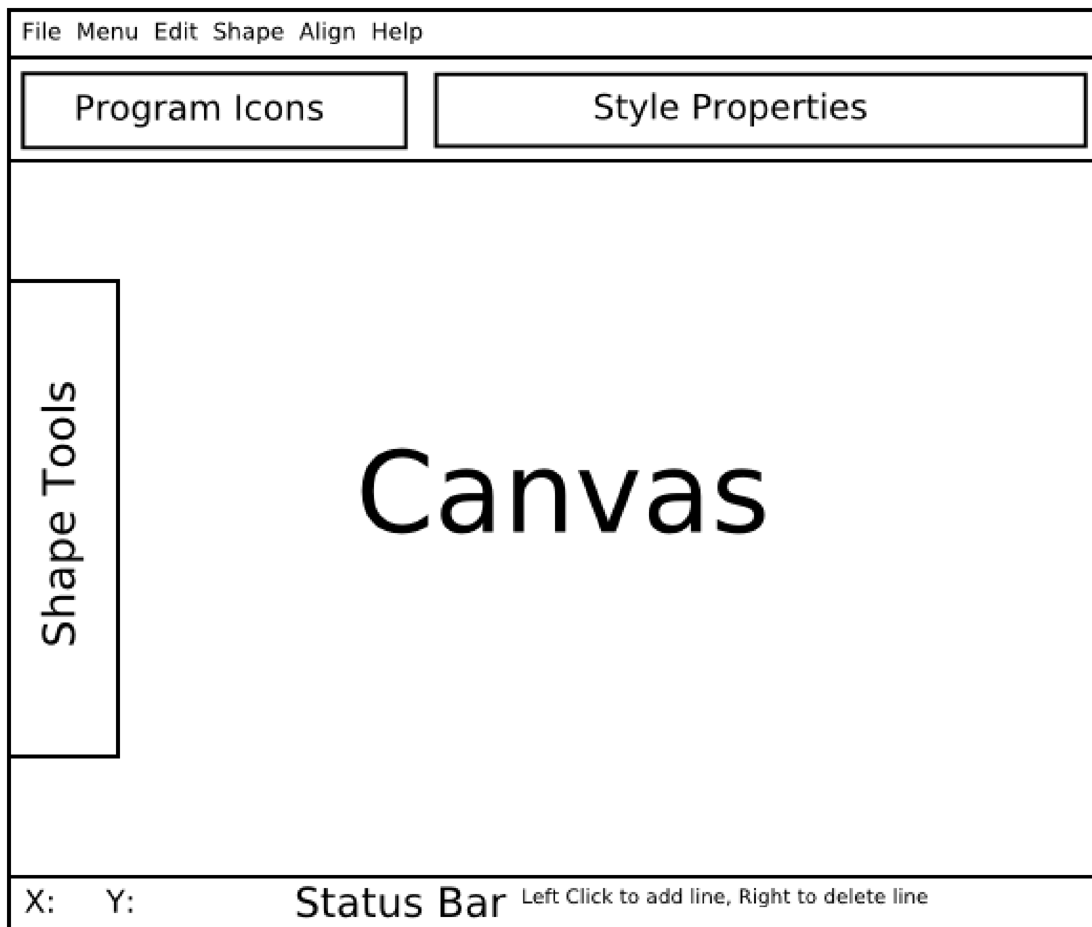


Figure 4.1: User interface prototype

The prototype shows a basic user interface. The interface includes the following:

- A canvas for drawing shapes onto
- Toolbars for selecting:
  - The shapes to draw onto the canvas
  - Styling options for the shapes
  - Options to load, save, and create a new diagram
- A menu which includes the default file, edit, and view menus. Additional menus are added for the more experienced users which include align options and changing the shapes properties

Every feature is accessible in one of these user interface features. A point of this design is that each feature in the requirements requires no user input (except for loading and saving). For expert users there is a property dialog box for shapes so that the properties of shapes can be fine tuned. Also all features are clearly labelled and are not hidden. Each button and menu item has a keyboard shortcut so that expert users can quickly access the feature that they want. The keyboard shortcuts are similar to what other programs have chosen

for instance the alignments menu options are the same as the alignment shortcuts on idraw. This allows a smooth transition for users that move from idraw to J-grams.

For beginner users there is a status bar which displays information regarding what the mouse buttons will perform when they are clicked. This is different from how xfig/jfig implement their help system as the mouse buttons functions have been standardised so that a left click is to add something, right to remove, and the middle mouse click is to finish drawing a shape. Some of these mouse functions only apply to a certain number of shapes but this assists beginner users to easily draw diagrams. This status bar also displays important information such as the dimensions of the current shape been drawn and the position of the mouse relative to the canvas.

This prototype was then created using a RAD style windows builder and shown to the test users for feedback. The main highlights of the feedback were:

- The interface was simple to use and catered for many types of user
- Users also wanted to easily draw specific diagrams such as ERD or UML diagrams
  - The shape queue feature in the secondary requirements will provide this functionality

Although not a complete usability analysis, this analysis has provided a user interface prototype which was accepted by a wide range of users. By targeting a wide range of users the interface will be well designed to suit many users and having a prototype to aim for during the implementation stage will make development much easier.

4.2 Class Diagram

A class diagram provides an abstract representation of how J-grams will look. It allows a system to be modelled as if it were being programmed but at an abstract level. It shows how objects are related to each other and what methods are accessible to other objects

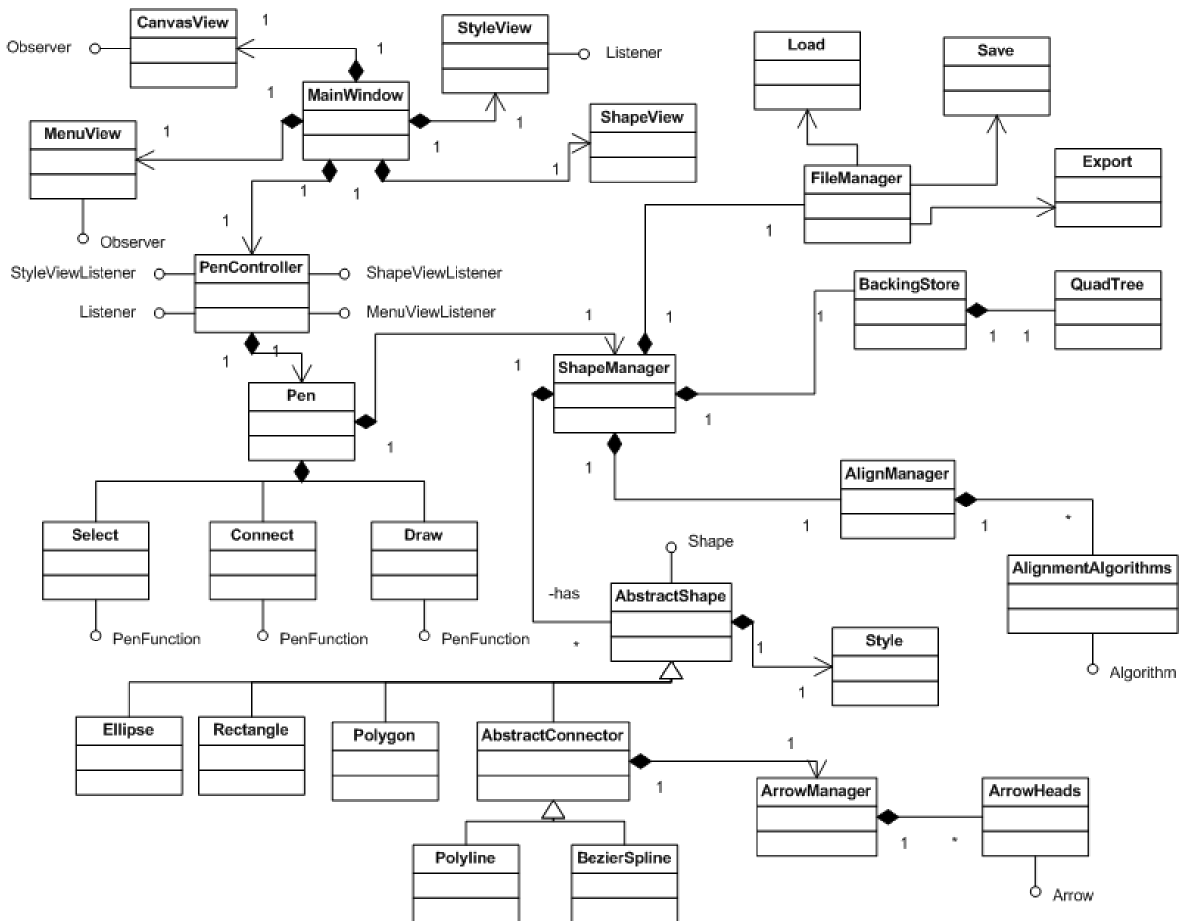


Figure 4.2: J-grams class diagram

The class diagram above shows a summarised view of the objects that make up J-grams. There are three parts that were diffcult design decision. The first is the Pen design which handles the different drawing styles required to draw shapes in the canvas. The second is the design of the shapes to support extensibility and the last, is how events flow in the system.

4.2.1 Pen

When a user interacts with the canvas via movement or clicking a button on the mouse a specific function must be called. These functions can range from selecting shapes, to drawing the actual shapes and modifying them. The design of J-grams includes a single pen, the pen is similar to that of a real world pen that has multiple functions for example a pencil that can draw and erase.

The pen controller is the global authority that receives all interface events such as menu clicks, button clicks, mouse movement and keyboard presses. The pen controller then dele-

gates these events to the required objects that are listening for a particular type of event.

A pen in J-grams is the middle layer between the user interaction with the canvas and the diagram itself. All mouse and keyboard events are passed to the pen so that they can change the diagram. At any time a pen can only have one function associated with it, and while that function is being used, all mouse and keyboard events are passed to it. The pen forms a type of event hierarchy where the events can first be processed in the actual pen to perform global functions before being passed onto a specific pen function which performs its specific function.

This design choice allows for different tools to be easily added to J-grams in the future. For example, drawing a rectangle is different from drawing a polygon as it requires multiple points being added to the shape rather than two. New pen functionality can be easily added by adding the appropriate function to the user interface, and adding the function to the pen controller.

4.2.2 Shapes

The shapes design uses the fact that most shapes have a lot of similarities with each other. This produces an abstract shape class which contains these similarities. By extending this class a shape is allowed to either use the default behaviour or its shape dependent behaviour. The shapes use polymorphism where most shapes are closely related to each other. For example, an ellipse and square are only separated by their draw and intersection functions.

All shapes defined extend the functionality of the abstract shape class, so each shape has a style associated with it. This style contains the fill, font and line properties and is used when the shape is drawn onto the canvas.

An extension of the abstract shape is the abstract connector which requires slight changes because of the differences between a normal shape and a connector. A connector can be a line, poly-line or a Bezier spline and these require multiple points to construct rather than the two points the abstract shape class supports. Further the connectors require an additional helper class to construct its arrow heads, and functionality to define which shapes it is connected to.

Custom shapes can easily be constructed using the composite shape that allows for shapes to be grouped. A change to the group will be reflected on to all of shapes contained inside the group. This class can be extended to allow users to add customised shapes that are diagram specific.

4.3 Design Patterns

Designing a large project is complex and some problems are more difficult to find a solution to than another. Design in computer science has been active for a long time, and there already exists solutions to most problems found in the design stage. This section describes the design patterns that were used to solve some of the design problems found in J-grams. It should be noted that design patterns are not a complete solution to the problem, rather they should be used as a guide to solving the design problem.

4.3.1 Model-View-Controller Pattern

The model-view-controller (MVC) design pattern is a design solution for separating the user interface from the actual content of the program. The model is the data of the program, while the view is the user interface. The view interacts with the model via the controller which facilitates the interaction between the two.

J-grams user interface is separated from the actual diagram stored in the model such as a change in the user interface does not affect the model. MVC solves the problem of how the user is supposed to interact with the model, by creating a controller which delegates user interface events to the required function in the model. This creates an event driven system that is easy to understand and be extended upon.

4.3.2 Observer Pattern

A problem during the design was how to notify the user interface when the model changes. For example when a shape is selected, the fill colour and line colour should change to what the selected shape has. Unfortunately, this problem was not solved during the design phase and proved to be a costly mistake. During the implementation phase it was found that the solution to this problem would require a lot of code to be rewritten to solve this problem.

The solution found, was to use the opposite of the MVC design pattern which is the observer pattern. This pattern registered observers which were interested in changes to the model. When changes occurred these observers would be notified of what change occurred. Java contains built in support for this type of pattern which allowed for an easy transition for the already written code. The interface Observer, and class Observable allow the pattern to be easily implemented.

In general the MVC pattern should always be paired up with the observer pattern.

4.4 SWT Library

The SWT library provides a platform independent programming environment that allows a Java user interface to take the native look and feel of the target platform that it is being run on. It makes heavy use of JNI features of java which accesses local methods on the platform to render the interface. The diagram editor uses the SWT library for drawing its user interface and using the canvas to actually draw the diagram on to the screen.

The canvas that SWT uses and which J-grams uses to draw the diagram onto is a vector graphics library called Cairo. Cairo is used in a lot of open source projects such as the GTK toolkit and Inkscape. Cairo can be thought of as the middle layer for all drawing operations in SWT, where these drawing operations are performed on different surfaces which can be seen as a piece of paper. Some surfaces included are XLib (the X windowing system) and Win32 (Windows) but more importantly as of version 1.2 of the library, it can now draw directly on to an SVG surface.

4.5 SVG Integration

There are currently two Java libraries that are capable of viewing, editing and creating SVG files, these are Batik and svgsalamander. J-grams needs a library which can open up an SVG file and be converted to the J-grams model of storage.

4.5.1 svgsalamander

Svgsalamander is a relatively new SVG library which supports most of the specification except the animation features. It was built because the other libraries namely Batik, were large and carried large dependencies when used. Due to its recent creation, some of its features are not documented very well, and it is hard to determine how to include it into a project.

Moreover it is difficult to understand how the library works, and how the components and structure are related to each other.

Svgsalamander uses the abstract window toolkit (AWT), specifically the Graphics2d functionality to render the SVG elements. This is not compatible with the SWT library. To move around this some type of compatibility layer must be added between when the shapes have been created in AWT and the diagram model of J-grams. This layer would retrieve all the properties of the AWT shape and convert it to the J-grams model so that it is easily drawn onto the SWT canvas.

4.5.2 Batik

Batik was created and now maintained by the Apache Software Foundation and supports all of the specification and parts of version 1.2. Batik is a very large and complex library and when included in a program adds a large dependency to it. It provides good documentation so that the structure and functionality of each component is easily understood and each function is also clearly documented in the API documentation.

Batik provides a core module which can manipulate, generate, create, convert, render and view SVG files. Batik also uses the AWT toolkit for drawing the shapes so as like before, another layer needs to be added which is much the same as the layer described for svgsalamander.

Batik is clearly the best choice for an SVG library to integrate with. The problem with svgsalamander is that it is opaque and monolithic while Batik is better structured and is more feature complete and compliant.

Chapter 5

Implementation

J-grams was implemented using the Eclipse IDE and the agile development methodology allowing changes to be made throughout the development lifecycle. J-grams itself is a large project of approximately 10,000 lines of code, with over 65 classes that interact each other to provide a solid foundation for further extension at a later date. This section describes some of the difficult implementation issues that were encountered during this phase and the solutions which J-grams uses.

5.1 Shape Intersections

When a connector is connected to the central connection point of a shape, the connector must find the intersection of itself with the shape it is connecting to. To the user this will appear as if the connector moves around the edges of the connected shape when it is rotated around it. This problem at first seems simple as finding the intersection with a line and a rectangle is easy, but as more complex shapes are added such as ellipses or rotated shapes this problem becomes more difficult.

A rotated ellipse is the most complicated as it introduces the following components into the standard ellipse equation:

$$
\begin{aligned}
x &= (x - a) \\
y &= (y - b) \\
x' &= xcos\theta + ysin\theta \\
y' &= ycos\theta - xsin\theta
\end{aligned}
$$

Solving the intersection with an ellipse and a line is done by simply inserting the line equation into the newly formed ellipse equation. This equation contains the translation and rotation formulas above. Solving it requires the use of the quadratic formula which will give two solutions which are then filtered to the desired range.

A generalised intersection library was created using the fact that all shapes are made up with a combination of move-to, line-to, cubic-to primitives. Although this library is not attached to J-grams at its current stage, it will be needed in the future once more features are added such as those defined in the secondary requirements. An ellipse can be easily constructed from two cubic curves and although it is not efficient it allows for connectors to find the intersection with complex shapes such as Bezier splines and Bezigons.

5.2 Quadtree

A diagram contains a list of 2-D shapes which are drawn onto a 2-D surface. When a user interacts with the diagram either by clicking on the canvas or moving the cursor over the canvas, the diagram needs to be searched to see if the mouse event was on any of the shapes. The need for an indexing structure that is capable of handling spatial data is required.

At first J-grams used a traditional way of storing the shapes which was in a list and iteratively searches the list to see if the mouse event was on top of the shape. Using this approach when there was a lot of diagrams on the canvas decreased the performance (˜100 shapes started to slow the system down). This structure provided $O(n)$ worst case performance and this did not include the cost of finding whether the mouse event intersected with the shape which at times was $O(n^2)$ (Bezier curves). For large diagrams a tree like structure is needed especially if they contain $>$100 shapes.

5.2.1 Structure

J-grams contains an implementation of a quadtree[3]. A quadtree recursively splits a bounded rectangular area into 4 sub-regions until a certain depth is reached which is defined when creating the tree. Each region contains a linked list which stores objects that belong to that region. To be added to a quadtree the object must occupy a rectangular area that intersects with the rectangular area of the root node.

As all shapes have a rectangular area they can easily be added into the quadtree. A shape is recursively inserted into the tree starting from the root node where it determines which region the rectangular area of the shape belongs in. At any level a shape can belong to more than one region. For example a square that is the full size of the canvas will be added to every region.

5.2.2 Performance

A quadtree is a tree structure that provides retrieval operations of $O(logn)$. Creating a quadtree from scratch is $O(4^d)$ where d is the depth of the quadtree. It adds a significant cost of starting J-grams up as it must create the entire quadtree as well as the linked lists inside each of the regions. An extension to J-grams that decreased this start-up time was to create the linked list only when the first shape was about to be added to a particular region. This optimization reduces the start-up time by half. Adding and removing objects from the quadtree is also expensive but these do not happen as much as searching and can safely be ignored at present. Although for real-time applications this would be a potential problem.

5.2.3 Memory Optimization

A quadtree is naturally memory intensive but the tradeoff in terms of retrieval performance out weights these problems. A possible memory optimization that may be explored in the future is to construct the quadtree dynamically instead of being created at runtime but this adds unneeded complexity to the structure. The quadtree implementation adds approximately one second to the start-up time with a depth of 6.

5.2.4 Zooming

A feature which is not supported by J-grams at the moment is zooming. A quadtree however provides the functionality to implement an efficient zooming algorithm. When the view is zoomed in on, the bounding box of the view is now smaller so only the shapes contained

inside of the new bounding box need to be displayed. As the quadtree splits up the area into smaller regions it is able to determine which shapes are contained inside a rectangular area and not just a single point. So a quadtree can find all the shapes that are inside of the smaller area and only display them. Scaling those shapes is implemented inside of the SWT library which support the scale transform.

5.3 Unbounded Quadtree

A problem with a quadtree is that it only works on a bounded rectangular area. Shapes which are added outside of the bounding box will not be added to the model and will not be displayed. As a diagram can be a very large size, a structure is needed which provides an unbounded area which shapes can be added to.

An extension of a quadtree that was created and what J-grams uses is called an unbounded quadtree. It allows for a virtually infinite amount of viewing area (memory restrictions come into effect though). By creating a type of grid that contains bounding boxes called views in each section of the grid. Each view contains a quadtree for that particular region. An unbounded quadtree provides a structure to manage all the views providing insertion, deletion, and retrieval functionality.

5.3.1 Structure

The internal structure of an unbounded quadtree is essentially a tree structure which contains a list of quadtrees at its nodes. The root node is the centre of the grid and contains pointers to four child nodes which are the surrounding corner views. The root node is a special case that contains an ordered linked list of all views that are directly above and below, and directly to the left and right.

Each node thereafter contains a single pointer to the next corner view. Inside each of these nodes is an ordered linked list pointing to the views directly to the left and below, right and below, left and above, or right and above. This depends on the views position in relation to the root node. Each node contains a rectangular area that it occupies which is used in the insertion, deletion and retrieval processes.

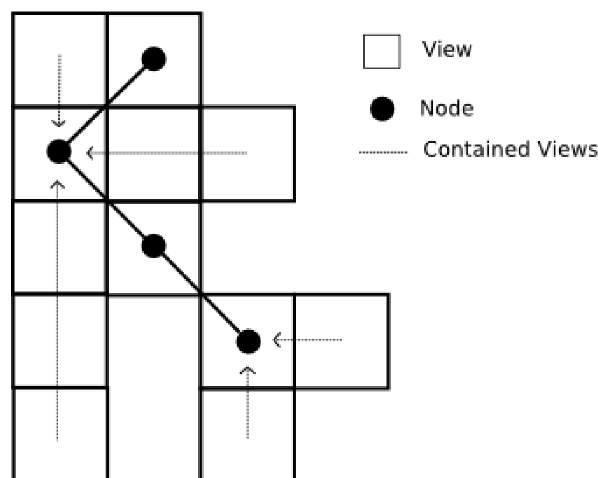


Figure 5.1: Unbounded Quadtree example

5.3.2 Example

When a shape is inserted outside of the initial view, the rectangle area of this new shape is used to iterate the tree. If the shape does not intersect any of the views contained inside the node, the node determines which corner view to move to next by seeing if the shapes area is greater then or less than the current nodes view. It recursively does this until an intersection occurs. When an intersection occurs the algorithm stops moving diagonally, and then starts to look at the views contained in that node. It calculates the view where it is going to be placed and creates this new view if it does not exist which in turn creates the quadtree for that view and the shape is inserted. A shape can intersect with more that one view so once an intersection has been found, the process starts again from its current place until an intersection cannot occur.

The unbounded quadtree still has all the efficiency of a quadtree but it now has additional complexity and requires more memory. An alternative which was discovered after implementing the unbounded quadtree was the R-tree[4] which is unbounded and uses less memory while still providing the efficient retrieval of a quadtree.

5.4 Backing Store

When a shape is drawn by the user it is added to the appropriate regions in the appropriate quadtree. A problem encountered was how to display the shape being drawn while displaying the actual diagram. SWT already provides double-buffering for its canvas which keeps two buffers of the image so that when a redraw event occurs the new buffer will only display when the redraw has been completed. This reduces the visible flicker that may be seen in some other programs when they are resized. Two solutions to this problem shall now be discussed, the first creates a lot of complexity and the second being the problem is ignored all together. The test results from these solutions are interesting, in that they are not what were expected especially for some test cases.

5.4.1 Double-Buffering

Drawing shapes onto a canvas is expensive and if you are redrawing the whole diagram every single time a mouse event occurs on the canvas, performance will start to decrease. This is noticed particular with more than 25 shapes when drawing another shape the rubber-banding effect of drawing the new shape does not keep up with the cursor. This also occurs using double-buffering that SWT provides. The problem is that every time the mouse moves it is causing a redraw event which then redraws all 25 shapes first and then draws the new shape. This redraw must be completed before the next mouse event occurs as if it does not it is queued until it is finished. This creates the noticeable lag when moving the cursor and the rubber-banding does not keep up with the cursor.

5.4.2 Triple-Buffering

A solution was created that used triple-buffering, which was another layer on top of the double-buffering SWT provided. The extra buffer was an image that was stored inside of a quadtree so that when shapes where added to the quadtree they were drawn onto the image. When a redraw event occurred only the image of the quadtree had to be redrawn. This adds complexity to the design however, as modifying or deleting a single shape invalidates the entire buffer and it must be redrawn internally. To enable a real-time view of the modification such as moving a shape the same problem will occur as previously discussed

because all shapes must be redrawn onto the buffer. An extension proposed was when the buffer is invalidated, the buffer would determine which rectangular area inside the buffer was invalidated and only update those shape inside this smaller area. Only a subset of all shapes would then be redrawn.

5.4.3 Performance Still Slow

The reason why this was not implemented is that the triple-buffer did not work as intended in the real world. In some test cases, in particular with Mozilla Firefox or any other program that had a lot of images contained inside of it, the performance of the triple-buffer would degrade. It could handle approximately 40 shapes but after this the lag would become too significant so that the program becomes unusable. No solution was found for this problem and so the triple-buffer was removed.

A compromise was created in the end. J-grams uses the double-buffering that SWT provides and it redraws all the shapes when a redraw event occurs but there is no caching. The performance problem was because anti-aliasing was turned on which smoothes the jagged edges on a shape. Turning this feature off allowed for more than 100 shapes to be drawn for every redraw event. The performance does eventually start to degrade but this is based on the speed of the system it is being run on so an option is provided to turn on or off anti-aliasing support.

5.5 Matrix Transformations

Implementing rotation and translating of a shape and zooming on the cavas requires the use of matrix transformations. The SWT library contains a transform method which can perform all the transforms needed on a shape. To utilise the functions provided by SWT, all shapes must be drawn onto the canvas centred at position (0, 0) because a rotate transform is rotated about the point (0, 0) so centring it makes a rotation of a shape simple. Once the rotation is complete a simple translate transform is used to move it to the correct position on the canvas. Internally a shape exists in the correct place all the time, the draw function of the shape just uses it current position to centre the shape at point (0, 0).

5.5.1 The Bounding Box of a Rotated Ellipse

Finding the bounding box of a shape is important for several reasons, the major reason being that the quad tree depends on the rectangular area the shape occupies. Finding the bounding box of rotated rectangle, line, or curve is trivial but finding the bounding box of a rotated ellipse is far more difficult.

5.5.2 Implicit Differentiation

Initially the idea was to use the ellipse equation from above, and differentiate it. Using this the maximum and minimum points are calculated, and where the derivative does not exist. The maximum and minimum points will give the hieght of the bounding box, while where the derivative does not exists will show position of the sides of the bounding box. The outcome was a complex equation that was difficult to understand involving a lot of testing which in the end led to nothing.

5.5.3 Matrix Transformations and a Rotated Ellipse

The solution was to use a path object in SWT to create an ellipse. A path object has a path data associated with it which is an array of x and y co-ordinates. It uses that data to construct complex arcs, cubics, or quad lines. With this knowledge the path object is able to calculate the bounding box of any path data given to it. However there seems to be an un-discovered bug in SWT that it will not properly transform a path. It seemed to be calling the correct method in the cairo library, but it would not overwrite the path data associated with that path. It seemed to be an error with the cairo library as the SWT JNI code was working correctly. The fix was to extract the path data from the path object and copy it to another newly created path data which is then rotated, and then the new path data is loaded into a new path object which calculates the bounding box of it.

5.6 SVG Support

As previously discussed a compatibility layer must be added onto the output of Batik which converts the AWT shapes to the shapes which are used in J-grams. Batik converts an SVG file and represents it internally as a document called a document object model (DOM) which contains elements and attributes. J-grams exploits this fact by recursively iterating through each element and getting its child elements if it has any. These elements are the primitives for drawing shapes in SVG for example 'rect' for rectangle and 'path' for a path. Once an element has been captured this element is then sent off to a factory inside of Batik which uses the elements name to return an AWT shape that can be used to draw on to an AWT canvas. Due to the similarities of the AWT and SWT shapes the conversion process is simple apart from the AWT path to SWT path conversion.

The conversion process takes a long time to complete and with some SVG files it can take approximately 8-9 seconds. This is because of the extra dependencies that must be loaded up during the execution of Batik. In particular are the Apache SAX parsers which parse XML files and the AWT component of Java. There is also a bottleneck when shapes are being added to the quadtree inside the J-grams model, as shapes are being added at a rate that is much faster than what a user could do which allows the inefficiency of adding to a quadtree to become evident.

5.6.1 Implemented SVG Parser Functionality

The compatibility layer can load most types of SVG diagrams. A list of features the layer supports is as follows:

- Support rectangles, circles, ellipses, squares, lines, paths and groups
- Fill and stroke styles
- Transformations including rotation

The layer cannot handle the more complex styling options which elements can have for example there is no support for having gradients or using a CSS style sheet to define the styles of elements. However because this layer is modular, extra functionality can easily be added at a later stage.

5.6.2 Extending the SVG Specification

Application dependent code such as storing what shapes a connector is connected to is handled by adding private attributes to elements that are part of the jgrams namespace. As SVG is an XML document, it allows XML namespaces to be added which contain uniquely named attributes and elements that can be added to a document. A requirement is that the namespace be declared before the use of any of these new attributes or elements are used. This namespace declares the following private attributes for elements:

- jgrams:connectStartShape = Element ID
  - The ID of the shape the start of the connector is connected to
- jgrams:connectStartPoint = left |right |top |bottom |central
- jgrams:connectEndShape = Element ID
  - The ID of the shape the end of the connector is connected to
- jgrams:connectEndPoint = left |right |top |bottom |central
- jgrams:connectStartStyle = Element ID
- jgrams:connectEndStyle = Element ID

This allows J-grams to load an SVG file and know what shapes are connected with one another. Due to the way all SVG compliant parsers work, they will simply ignore these private attributes and render the file as described by the SVG file which will be the diagram. Allowing the inclusion of private elements and attributes is a powerful feature and allows for easy extension in the future as parsers which do not understand the namespace simply ignore it. A possible extension could be to allow the arrow head to be any shape which is feasible already as the SVG file just points to a normal element instead of the pre-programmed ones J-grams contains.

As the layer interacts with Batik the DOM of the SVG file is not kept intact and when a file is opened and then saved, the saved file will be completely different from what it originally was, but will still render the same diagram. This is because when saving J-grams does not use Batik. J-grams iterates through the shapes that are selected to be saved and calls a toSVG() method on each which returns a string representation that contains the SVG element to be inserted into the file. This is a problem that will need to be addressed in the future and would mean that J-grams would need to be more tightly integrated with Batik.

5.7 Creating Custom Widgets for SWT

The SWT library allows for the construction of custom widgets, such as buttons which can contain an image. This is used to construct a colour picker similar to that seen in the Microsoft Office suite which displays a common list of colours when clicked and also allows a user to select a colour using the colour chooser dialog which displays all the colours available.

5.7.1 Extending SWT

A custom widget is implemented by extending the canvas class. A SWT canvas can receive all user interactions from the system which means it can be notified when a user clicks on

it. The underlying feature to construct a colour picker is to construct a shell which in SWT is a window which displays information. A shell then contains the canvas which we can draw on, which is similar to the actual drawing of diagrams on the canvas. This difference between this shell and others is that it is borderless, and does not steal the focus from the main window.

5.7.2 Displaying and Hiding the Colour Picker

As the shell is borderless all that is displayed is the canvas. Determining whether a mouse event occurs on this canvas is simply done by listening for mouse events in the new shell. A problem occurs when a user clicks outside of the canvas which has the effect of closing the colour picker when it is open. To implement this in SWT required listening to all mouse events in the system and determining whether the mouse event was outside of the colour pickers display area. When the canvas is displayed it does not steal the focus from the window calling it, so that when it is clicked and the colour picker is displayed the diagram is still modifiable as it still retains focus. This is a user interface feature to keep the workflow contained into a single window. Most freely available colour pickers for SWT are both slow and do not have this feature.

The end result is the following widget:

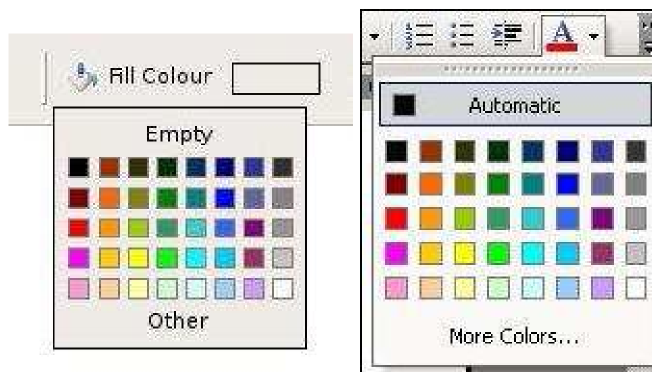


Figure 5.2: Left: Implemented SWT colour picker Right: Microsoft Office colour picker

It is very similar to the Microsoft Office design. There are a few improvements that could be added. The first being that when a user moves the mouse over Fill Colour there is no visual recognition that this is a button, it does not display a button outline. The second is that the popup window does not look like it is attached like the Office counterpart does.

Chapter 6

Conclusions

This project has designed and implemented a functional diagram editor called J-grams. The core requirements have been implemented and J-grams is capable of creating a wide range of diagrams. The implementation of J-grams has created a solid framework which will be extended later to include the secondary requirements. These requirements will move J-grams more in line with large diagram editors such as Microsoft Visio and provide capabilities to draw specific types of diagrams.

During the development of J-grams a lot has been learnt including better project management skills, and design. The implementation included a quadtree, and a extension to a quadtree called an unbounded quadtree. Libraries were created such as a generic intersection library for shapes, and creation of new widgets for SWT. Some of design issues that were found can be extended to solve problems in other software engineering projects

6.1 Future Work

J-grams is a fully functioning diagram editor now that the primary requirements have been implemented. It is now time to move onto the secondary requirements that shifts J-grams from drawing simple and generalised diagrams to handling more specialised diagrams that are used in the real world.

6.1.1 R-Tree

The R-tree is a better implementation than a quadtree, and it is unbounded. It provides a more efficient storage system and has the structure of a tree so retrieval is again O(log n). Future work will involve changing the quadtree implementation to the R-tree algorithm.

6.1.2 Shape Queue

The shape queue is the most interesting feature in the secondary requirements and more analysis needs to be performed on this to see whether it is good feature in terms of usability. The shape queue is going to implement the functionality that the commercial diagram editors provide allowing specific diagrams to be constructed.

6.1.3 Animation Support

During a brief interview with some users about the usability of the interface a question was raised asking what features they would like to see. The most important feature asked for was animation support. This was not listed in the secondary requirements as it is would

require a lot of backend changes but is worth mentioning for future research. The uses for animation were focused on one idea which was capturing the changes applied on the canvas and turning that into an animation. For example some ideas were visualising how an array works by changing the fill colours of several rectangles aligned horizontally, or by moving a solid rectangle inside one of those rectangles. This feature would be particularly helpful for people doing presentations or lecturers.

Bibliography

[1] Dia homepage. http://www.gnome.org/projects/dia/ [Online; accessed 11-October-2006].

[2] Jfig homepage. http://tams-www.informatik.uni-hamburg.de/applets/jfig/ [Online; accessed 1-October-2006].

[3] FINKEL, R. A., AND BENTLEY, J. L. Quad trees a data structure for retrieval on composite keys. *Acta Informatica 4*, 1 (March 1974), 1–9.

[4] GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD International Conference on Management of Data* (1984), 45–57.

[5] IVTOOLS. idraw homepage, 2006. http://www.ivtools.org/ivtools/idraw.html [Online; accessed 13-October-2006].

[6] MICROSOFT. Microsoft visio 2003 homepage. http://office.microsoft.com/visio/ [Online; accessed 11-October-2006].

[7] MLLER, H.-G. udraw homepage. http://www.informatik.uni-bremen.de/~davinci/ [Online; accessed 11-October-2006].

[8] SUPOJ SUTANTHAVIBUL, B. V. S., AND KING, P. Xfig homepage. http://epb.lbl.gov/xfig/ [Online; accessed 13-October-2006].

[9] YWORKS. yed homepage. http://www.yworks.com/products/yed/ [Online; accessed 11-October-2006].