Porting a TreeMap Library from AWT/Swing to SWT/JFace

Lirim Jusufi
Abstract
Hierarchical structures are commonly used for storing data and information. Especially in computer science, the state of the art are still hierarchical file systems storing data in treelike containment structures having the two elements: files and folders. The increasing storage capacities and quantities of data make it more and more complex to keep track of where and data are stored, since the data structures grow both in depth and breadth. E.g., finding with one look large, potentially unnecessary, files that could be deleted is not with the current graphical file system browsers like the Windows Explorer. It requires command-line tools or search dialogs to identify files in question, which are usually represented in list or tree form.

Over the years, many visualisation techniques have been developed that can help people have a better overview about the hierarchical data. They allow seeing the single elements in context and proportion to the other elements in the hierarchy.

One of the most prominent techniques for visualising hierarchical data are Treemaps which were first presented by Ben Shneiderman in 1991.

In spite of VizzAnalyzer being a flexible tool for software analyzing, it still lacks a good overview about the containment hierarchy. Expanding it with a treemap visualisation technique will help in solving this problem. Since VizzAnalyzer is integrated into Eclipse as a plug-in, it is in our interest for treemap viewer to be implemented in SWT GUI so it can be integrated more easily into the VizzAnalyzer.

This bachelor thesis examines the way of making the implementation of a treemap viewer integrable into the VizzAnalyzer.

Keywords: Hierarchical structure, containment hierarchy, treemap, VizzAnalyzer, SWT.
## Table of Contents

1. Introduction .................................................................................................................. 1  
   1.1 Problem .................................................................................................................. 1  
   1.2 Goals and Criteria ................................................................................................. 2  
   1.3 Motivation .............................................................................................................. 2  
   1.4 Outline ................................................................................................................... 3  

2. Background .................................................................................................................... 4  
   2.1 Grail ....................................................................................................................... 4  
   2.2 Treemaps ............................................................................................................... 4  
      2.2.1 Ordered Treemaps ......................................................................................... 4  
      2.2.2 Squarified Treemaps ............................................................................... 4  
      2.2.3 Strip Treemaps .......................................................................................... 4  
      2.2.4 Quantum Treemaps .................................................................................. 5  
   2.3 Widgets .................................................................................................................. 5  
   2.4 AWT ...................................................................................................................... 5  
   2.5 Swing .................................................................................................................... 5  
   2.7 JFace .................................................................................................................... 6  
   2.8 Eclipse .................................................................................................................. 6  
   2.9 Eclipse Plug-ins ................................................................................................... 7  

3. Requirements and Architecture .................................................................................. 8  
   3.1 Requirements ......................................................................................................... 8  
   3.2 Architecture .......................................................................................................... 9  
      3.2.1 Architecture of the Treemap Library ............................................................ 9  

4. Implementation and Evaluation .................................................................................. 10  
   4.1 Package “project.treemapSWT” ......................................................................... 10  
      4.1.1 TMAction Class ......................................................................................... 10  
      4.1.2 TMAlgorithm Class .................................................................................... 10  
      4.1.3 TMComputeDraw and TMComputeDrawAdapter classes ....................... 12  
      4.1.4 TMComputeSize and TMComputeSizeAdapter classes ....................... 12  
      4.1.5 TMEvent* classes ...................................................................................... 13  
      4.1.6 TMExceptionBadTMNodeKind class ........................................................... 13  
      4.1.7 TMExceptionLeafTMNode class ................................................................. 13  
      4.1.8 TMExceptionNullParameter class ............................................................. 13  
      4.1.9 TMExceptionUnknownTMNode class .......................................................... 14  
      4.1.10 TMModelNode class ................................................................................ 14  
      4.1.11 TMModelUpdater class .......................................................................... 14  
      4.1.12 TMModelUpdaterConcrete class .............................................................. 14  
      4.1.13 TMNode class ............................................................................................ 15  
      4.1.14 TMNodeAdapter class ............................................................................. 16
4.1.15 TMNodeEncapsulator class ........................................... 16
4.1.16 TMNodeModel class ................................................. 16
4.1.17 TMNodeModelComposite class .................................... 16
4.1.18 TMNodeModelRoot class ........................................... 16
4.1.19 Thread classes ....................................................... 16
4.1.20 TMThreadModelClass ................................................ 16
4.1.21 TMThreadChangeDrawing Class .................................. 17
4.1.22 TMThreadChangeSizing ............................................ 17
4.1.23 TMThreadLostChild ............................................... 17
4.1.24 TMThreadNewChild ............................................... 17
4.1.25 TMThreadTreeBuilder ............................................. 17
4.1.26 TMThreadLock ..................................................... 17
4.1.27 TMThreadQueue .................................................... 17
4.1.28 TMThreadUpdateDraw ............................................ 17
4.1.29 TMThreadUpdateSize ............................................ 17
4.1.30 TMUpdater Class .................................................. 17
4.1.31 TMUpdaterConcrete Class ....................................... 18
4.1.32 TMView Class ..................................................... 18
4.1.33 TreeMap Class ..................................................... 18

4.2 Package “project.treemapSWT.graphViewer” ....................... 19

4.2.1 Demo Class ......................................................... 19
4.2.3 Config Class ...................................................... 19
4.2.4 Graph Class ....................................................... 19
4.2.5 TMGraphDraw Class ............................................. 19
4.2.6 TMGraphModelDraw Class ..................................... 19
4.2.7 TMGraphNode Class ............................................. 19

4.3 Implementation Details ............................................... 19
4.4 Evaluation .............................................................. 23

5. Conclusion and Future work ........................................... 25
  5.1 Conclusion ........................................................... 25
  5.2 Future Work ........................................................ 26

6. References ................................................................... 27

Appendices ...................................................................... 28

Appendix A – Package Structure and libraries of Treemap Viewer .... 28

A.1 Package project.treemapSWT ......................................... 28
A.1.1 TMAction class ..................................................... 29
A.1.2 Algorithm classes ................................................. 29
A.1.3 Compute classes ................................................... 29
A.1.4 EventUpdate classes .............................................. 30
A.1.5 Exception classes .................................................. 30
A.1.6 Model classes ....................................................... 30
List of Figures

Figure 2.1 ......................................................................................................................... 6
Figure 2.2 ......................................................................................................................... 7
Figure 3.1 ......................................................................................................................... 9
Figure 4.1 ......................................................................................................................... 10
Figure 4.2 ......................................................................................................................... 11
Figure 4.3 ......................................................................................................................... 12
Figure 4.4 ......................................................................................................................... 12
Figure 4.5 ......................................................................................................................... 13
Figure 4.6 ......................................................................................................................... 13
Figure 4.7 ......................................................................................................................... 13
Figure 4.8 ......................................................................................................................... 13
Figure 4.9 ......................................................................................................................... 14
Figure 4.10 ....................................................................................................................... 14
Figure 4.11 ....................................................................................................................... 15
Figure 4.12 ....................................................................................................................... 16
Figure 4.13 ....................................................................................................................... 17
Figure 4.14 ....................................................................................................................... 18
Figure 4.15 ....................................................................................................................... 18
Figure 4.16 ....................................................................................................................... 18
Figure 4.17 ....................................................................................................................... 19
Figure 4.18 ....................................................................................................................... 20
Figure 4.19 ....................................................................................................................... 20
Figure 4.20 ....................................................................................................................... 21
Figure 4.21 ....................................................................................................................... 21
Figure 4.22 ....................................................................................................................... 21
Figure 4.23 ....................................................................................................................... 21
Figure 4.24 ....................................................................................................................... 22
Figure 4.25a ..................................................................................................................... 22
Figure 4.25b ..................................................................................................................... 23
Figure 4.26 ....................................................................................................................... 23
Figure 4.27 ....................................................................................................................... 24
Figure 4.28 ....................................................................................................................... 24
Figure A.1 ......................................................................................................................... 28
Figure A.2 ......................................................................................................................... 29
Figure A.3 ......................................................................................................................... 29
Figure A.4 ......................................................................................................................... 31
Figure A.5 ......................................................................................................................... 32
Figure A.6 ......................................................................................................................... 32
Figure A.7 ......................................................................................................................... 33
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWT</td>
<td>Standard Widget Toolkit</td>
</tr>
<tr>
<td>AWT</td>
<td>Abstract Window Toolkit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>GML</td>
<td>Graph Modelling Language</td>
</tr>
<tr>
<td>GRAIL</td>
<td>Graph Implementation</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JDT</td>
<td>Java Development Tools</td>
</tr>
<tr>
<td>RCP</td>
<td>Rich Client Platform</td>
</tr>
</tbody>
</table>
1. Introduction

Hierarchical structures are commonly used for storing data and information. Especially in computer science, the state of the art are still hierarchical file systems storing data in treelike containment structures having the two elements: files and folders. The increasing storage capacities and quantities of data make it more and more complex to keep track of where and data are stored, since the data structures grow both in depth and breath. E.g., finding with one look large, potentially unnecessary, files that could be deleted is not with the current graphical file system browsers like the Windows Explorer. It requires command-line tools or search dialogs to identify files in question, which are usually represented in list or tree form.

These problems are not only limited to everyday applications like organizing files on a computer, but also in more advanced situations like the development of software systems. Software projects are usually also hierarchically structured. Starting with the organization of source and binary files in projects and modules, extending to the hierarchical structure in modern object-oriented programs describing packages containing files and classes, which in turn contain fields, methods, blocks and statements, etc., maybe ending with the inheritance hierarchy of classes and interfaces. These increasingly complex software structures are visualized by the same graphical means as files and folder, i.e., lists and trees. Thus they inherit the same limitations of visually viewing the entirety of a hierarchy and the involved elements. Limitations include not being able to compare the elements in their proportion according to one or more properties.

The VizzAnalyzer is a software analysis tool developed at the Växjö University. It is used for extracting information from various representations of software systems, including source code, byte code and other design specifications. Besides calculating software metrics, it is used for visualizing the design and architecture of the analysed systems [1].

1.1 Problem

Over the years, many visualisation techniques have been developed that can help people have a better overview about the hierarchical data. They allow seeing the single elements in context and proportion to the other elements in the hierarchy.

The objective of this thesis is to extend the VizzAnalyzer with new visualization capabilities, allowing displaying and selecting hierarchically structured data in an efficient way. User should be able to find the necessary information more easily and faster.

A pre study, not being part of this thesis has already selected treemaps as a suitable visualization. See Section Background for details about treemaps. The treemap algorithm has been implemented in a freely available Treemap Library [2]. It offers a default implementation for analysing and displaying files and folders of a file system. Since this library is implemented using AWT/Swing, but the VizzAnalyzer is integrated into the Eclipse framework being based on SWT/Jface, it is necessary to port the implementation, so that it can be used to extend the VizzAnalyzer.

Therefore, the problem addressed by this thesis is to port the Java Treemap Library implemented by Bouthier [2]. Furthermore, it needs to be extended to be able to visualize information provided as graphs of the Grail library.
This is a non-trivial problem to solve, since the used APIs have quite some differences. Additionally, the underlying design differs regarding the handling of threats. Solving this requires a deep understanding of both technologies.

In future work, the ported library would be used on implementing a plug-in for Eclipse platform which helps visualising file systems and containment hierarchy of software system into a treemap.

1.2 Goals and Criteria
The above outlined problem is solved, when the following goals are reached. Each goal is defined by a number of criteria, which can be used for assessing if the goals are met.

- The first goal is to write an algorithm which reads directory structures from file system and convert them to a graph using the Grail library. This goal is reached if graph is properly written to “*.gml” file representing the underlying files and folders of a file system together with some selected properties. This is important for test data generation and provides the future interface for the VizzAnalyzer.
- The second goal is to adjust the Treemap Library so it can read graph structures stored in a “*.gml” file and display them into a treemap. This goal is reached if the hierarchy is displayed properly in the treemap.
- The third goal of this thesis is to understand the differences and similarities between Java AWT/Swing and SWT/JFace. This part of incremental and iterative porting process is mainly focused on research.
- The fourth goal is to understand the implementation of the Treemap library. Divide the classes on which ones should be ported and which not. Understand the dependencies among them. This is also focused on research.
- The fifth and the final goal is porting the whole application from AWT/Swing to SWT/JFace. This goal is reached if application is compiled without any errors and if results displayed are identical as the results of the application implemented in AWT/Swing. Thus the features of the original application shall be fully preserved.

1.3 Motivation
The VizzAnalyzer is a flexible software analysis framework. It consists of various front-ends for fact extraction from source code and other program representation. Currently the selection of files and folders being input to the front-ends is not very intuitive and lacks overview. The user has either to specify file by file or create an eclipse project. Furthermore the extracted information can be visualised in 2D/3D using various tools, but none of them allows to get and good overview about the containment hierarchy [1].

This problem can be solved by using already developed visualising techniques and integrate them into the application. Treemaps are oldest and most prominent visualization method when it comes to representation of containment hierarchy.
1.4 Outline
The structure of this thesis is as following:
Chapter 2 gives some information about the Grail, Treemaps, AWT, Swing, SWT and JFace in general. Chapter 3 provides a list of existing main features of the application and additional features of the ported application. Chapter 4 gives information about the current architecture of the application. Chapter 5 gives some information about implementation, general description of the program code, some code fragments, look of the program etc. Chapter 6 concludes the thesis and gives information about the future work.
2. Background

This section describes the technologies related to the problem. It gives a brief description of Grail, Treemaps, SWT/JFace, AWT/Swing and Eclipse platform and plug-ins.

2.1 Grail
Grail (GRAph ImpLementation) is graph library developed at the MSI – Växjö University. It is an API (Application Program Interface) for storing and manipulating binary relations – data storage. Basically any kinds of attributes are allowed [11]. There are three types of elements which are fundamental to Grail: Node, Edge and Graph. These elements can be manipulated by the algorithms stored in the library.

2.2 Treemaps
Treemaps were first presented by Ben Shneiderman in 1991. They are a space-filling visualisation method designed for representing hierarchical data[3]. Hierarchy is recursively mapped to rectangles where each rectangle represents a particular data. In case of a file system, each rectangle represents either a directory or a file. In order to maintain the hierarchy, every child’s rectangle is presented within the parent’s rectangle.

There are different algorithms for laying out the treemaps, but the original algorithm came from Sheiderman. It uses parallel lines, horizontal or vertical, for dividing the rectangles, and due to high aspect ratios it produces long thin rectangles. This algorithm is called “Slice and Dice”[3].

Beside the “Slice and Dice” there are other layout algorithms, such as:

- Ordered Treemaps
- Squarified Treemaps
- Strip Treemaps
- Quantum Treemaps etc.

2.2.1 Ordered Treemaps
This algorithm creates a layout in which items that are next to each other in the given order are adjacent in the treemap. This is important if we need to preserve some kind of order among visualised objects. Even though it is not as simple as the Slice and Dice algorithm, it provides much better layout[4].

2.2.2 Squarified Treemaps
Because of high ratio aspect standard algorithms produce thin rectangles which are not good for laying out large hierarchical structures. Because of its better layout, Squarified treemap algorithm is used for visualising large hierarchical structures, such as directory structures [5] etc.

2.2.3 Strip Treemaps
This is modified version of the Squarified treemap algorithm. It is also used for visualising hierarchical data such as directory structures. The algorithm works by processing input rectangles in order than laying them out in vertical or horizontal strips of varying thickness [6].
2.2.4 Quantum Treemaps
This layout algorithm is used to present clusters of visual information such as images[6].

In this implementation of the treemap library there are two available algorithms, “Classic” which in fact is “Slice and Dice” algorithm, and “Squarified” or “Squaring” algorithm.

2.3 Widgets
A widget is an element of GUI (graphical user interface), it can be a button, scrollbar, area for text editing etc. Some of the widgets can be interactive for example buttons; others can act as containers that group other widgets for example windows, panels, tabs etc [12].

A widget toolkit of GUI toolkit is a set of widgets for use in designing applications with GUI (graphical user interface). The toolkit besides that provides the widgets it also handles the user events, for example clicking on a button. When an event is detected it is passed onto the application where it is dealt with.

There are different widget toolkits for example GTK+ used by GNOME applications, SWT - Standard Widget Toolkit, Swing – java widget toolkit, AWT – abstract window toolkit, JFace etc.

2.4 AWT
AWT (Abstract Window Toolkit) is Java's original platform-independent graphical user-interface widget toolkit. It is a bridge to the underlying native user interface. It provides a basic set of GUI widgets such as buttons, text boxes and menus. It also provides the AWT Native Interface, which enables rendering libraries compiled to native code to draw directly to an AWT drawing surface. AWT also provides some miscellaneous useful utilities that can also be used by Swing such as: Access to the system tray on supporting systems and the ability to lunch some desktop applications such as web browsers and email clients from a java application [7].

2.5 Swing
Swing is a widget toolkit for Java. It is a part of Sun Microsystems’s Java Foundation Classes (JFC) – an API for providing a graphical user interface (GUI) for users. Swing was developed to provide more sophisticated set of GUI components than AWT (Abstract Window Toolkit). But we shouldn’t understand it as a replacement for AWT. Actually Swing is built on top of the core AWT libraries. Because Swing does not contain any platform-specific (native) code, you can deploy the distribution on any platform that implements the Java Virtual Machine. In order for Swing classes to be available for use JDK should be installed on the platform [7].

Swing is not an acronym. The name actually is a collaborative choice of its designers when the project was first introduced in late 1996. And it’s in development since. When the first version of Swing was released the libraries contained nearly 250 classes and 80 interfaces.
2.6 SWT
Although many of us refer to SWT as toolset, it is actually a software library which consists of packages that contain Java classes and Interfaces.

SWT (Standard Widget Toolkit) is UI toolkit developed by IBM and used by Eclipse platform and most other Eclipse projects. It is implemented by creating thin native wrappers for the underlying operating system’s user-interface API. The bulk of SWT’s source is Java code, which defers as much work as possible to the appropriate operating system native. Thus, when you create a tree widget in SWT, it calls through to the operating system to create a native widget tree [8]. The result is that the SWT applications tend to look and behave exactly as native applications on the system they’re running on. No java emulation is done.

2.7 JFace
JFace is a UI toolkit that provides helper classes for developing UI features that can be tedious to implement. It is a layer that sits on top of widget system, and provides classes that handle common UI programming tasks. It brings model view controller to SWT (Standard Widget Toolkit) [8].

2.8 Eclipse
Eclipse is a Java-based, extensible open source development platform. By itself, it is simply a framework and a set of services for building a development environment from plug-in components. While different plug-ins can be developed, Eclipse RCP (Rich Client Platform) comes with some already installed plug-ins such as JDT (Java Development Tools), which turns Eclipse RCP into Java IDE (Integrated Development Environment). Thanks to Eclipse’s Extensible architecture it allows the developers to integrate their own tools [9].

![Figure 2.1 Simplified Eclipse Architecture](image)

Figure 2.1 Simplified Eclipse Architecture [9]
2.9 Eclipse Plug-ins

A plug-in is a component that provides a certain type of service within Eclipse. Each plug-in is embodied in a class which is called plug-in class. This class provides configuration and management support for plug-ins. Every eclipse installation includes some plug-ins installed. Every plug-in is installed in its own plug-in folder. A plug-in is described by a manifest file which is in XML format, this file is called plug-in.xml and it is located in plug-ins folder. This file tells Eclipse what is necessary to activate the plug-in. Plug-in instances are managed by Eclipse runtime and they are not constructed by application programs [10].

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?Eclipse version="3.2"?>

<plugin>
  <extension id="application"
    point="org.Eclipse.core.runtime.applications">
    <application>
      <run class="exampleplugin.Application"></run>
    </application>
  </extension>
</plugin>
```

Figure 2.2 Basic plugin.xml file

To install a plug-in in Eclipse, all the files that constitute the plug-in (the manifest file, jar files etc.) must be copied into a single folder under the installation’s plug-ins directory. Such a plug-in can later be activated by the Eclipse runtime. And by activation it means loading its runtime class and instantiating and initializing its instances.
3. Requirements and Architecture

In this chapter we will show some basic requirements of this thesis, and say something about the architecture of the implementation in general.

3.1 Requirements

This section will give the list of some basic features of the original implementation and new functional requirements of the ported application.

As mentioned before the objective of this thesis is porting an already implemented library from AWT/Swing to SWT/JFace. So, the basic features of the ported application will not be different from the features of the original one. This is actually the main requirement, to preserve the functionality.

Anyway, we provide a list of the existing main features of the original implementation:

- **Read file systems** – this feature allows the reading of file systems (files and directories) and display the each read element in the treemap.
- **Layout algorithm** – this feature allows the user to change the layout algorithm of the treemap. In other words, it changes the way the treemap is displayed.
- **Border Size** – this feature allows the user to adjust the size of the borders surrounding the nodes. By size of the border we mean the thickness of the line.
- **Tooltip text** - is a GUI element which appears when mouse hovers over a particular node or other element of the treemap and displays information about that element.
- **Resize** – the treemap should be resizable, it means that when the main window that displays the treemap is resized, the treemap should be resized automatically itself.

These features will be available also in the ported version. In addition the following the features will be available:

- **Read file systems** – read files and folders are saved as graph in a gml file.
- **Read graphs** – the program should be able to read the directory structures stored as graph in a gml file and display them in the treemap
- **SWT/JFace GUI** – the whole implementation should be ported using SWT/JFace.
3.2 Architecture
Since Java Treemap Library is going to be an Eclipse plug-in it is important that application is implemented in SWT/JFace.

3.2.1 Architecture of the Treemap Library
The architecture of the Treemap Library does not change due to the porting actions. Changes affect only the implementation on statement and method level. This means that basically the architecture remains the same. The overall architecture can be seen in fig.4.1.

The single components and their purpose are described in the following: GML and Graph are responsible for reading the graphs stored in “*.gml” files. And when necessary they can be used for creating new “*.gml” files from directory structures. TreeViewer is the main part of the application. It also contains some information about how the nodes should be drawn. TreeView API is responsible for all the calculations (node size and layout calculations) and drawings, managing application threads etc. Treemap is the part where the results from all the calculations and the drawings are displayed, the part where the treemap is shown.

So basically after the graph is read from the “*.gml” file it is passed to the TreeView API through the main part for calculating and drawing. After all the calculation and drawings are done it sends the results back to the main part which later displays the Treemap.

After analyzing the structure of the original implementation and comparing AWT/Swing to SWT/JFace it became clear that no changes to the high level of the application are necessary. The architecture of the application would not change due to porting from Swing to SWT/JFace.
4. Implementation and Evaluation

The previous chapter described the architecture of the Treemap Viewer. This chapter will focus on design and implementation. After discussing the classes and design in general we will continue discussing the implementation of Treemap Viewer together with selected code parts that were not straight forward to port.

Treemap Viewer consists of two packages `project.treemapSWT` and `project.treemapSWT.graphViewer`. Each package contains different number of classes, which are described in the following.

4.1 Package “project.treemapSWT”

This package contains the general Treemap Library classes which are used for calculating layout algorithms, drawing and manipulating the treemap. See appendix A.1 for more information.

4.1.1 TMAction Class

`TMAction` class manages the action on the treemap. It implements MouseListener and MouseTrackListener in order to enable mouse actions like clicking and hovering.

![Figure 4.1 TMAction class](image)

`mouseUp()` method is called when mouse button is released. It uses the coordinates of the current position of the cursor to zoom over the nodes by calling the `zoom()` method. If shift is pressed the `unzoom()` method is called.

`mouseHover()` method is called when cursor stands over the treemap for short time. ToolTip text is displayed when this method is called.

4.1.2 TMAlgomath Class

The `TMAlogoth` abstract class represents the algorithm of the treemap. It should be subclassed by every class that wants to implement a particular treemap such as Classic and Squarified. A subclass can also override the `drawNode()` method to have a customized drawing of a node.

- `draw()` method starts the process of drawing the treemap.
- `drawNodes()` method draws the node and recourses the drawing on its children.
- `drawChildren()` method draws the children of a node, by setting the area first, dependant on the algorithm used.
- `fillNode()` method fills the nodes with color.
- `drawNodesTitles()` method draws the nodes titles.
- `switchAxis()` method switches the axis on how the nodes should be drawn, vertically or horizontally, and returns a new axis.

The two subclasses that extend `TMAlgorithm` class are `TMAlgorithmClassic` which implements a classic treemap drawing algorithm and `TMAlgorithmSquarified` which implements squarified treemap drawing.

![Diagram of TMAlgorithm, TMAlgorithmClassic and TMAlgorithmSquarified classes](image)

Figure 4.2 TMAlgorithm, TMAlgorithmClassic and TMAlgorithmSquarified classes
4.1.3 TMComputeDraw and TMComputeDrawAdapter classes

*TMComputeDraw* interface should be implemented by every class that implements algorithms for drawing TMNode. The methods to implement are: `getFilling()`, `getToolTip()`, `getTitle()` and `getTitleColor()`.

*TMComputeDrawAdapter* class implements an adapter for the *TMComputeDraw* interface for users of the TMModelNode interface.

4.1.4 TMComputeSize and TMComputeSizeAdapter classes

*TMComputeSize* interface should be implemented by every class that implements the algorithm for computing the size of *TMNode*. Computing of the size depends on the kind of *TMNode*, so in order to avoid any errors for incompatibility the kind of the given *TMNode* should be tested and if incompatible it should throw a *TMExceptionBadTMNodeKind*.

*TMComputeSizeAdapter* class implements an adapter for the TMComputeSize interface.
4.1.5 TMEvent* classes

`TMEventUpdate` abstract class implements an update event send by a `TMUpdateConcrete` to the TMNodeModelRoot. Such an event is sent when something is has changed in TMNode. The `TMEventUpdate` should be subclassed to indicate the type of event to send.

`TMEventUpdateDraw` class implements an update draw event.

`TMEventLostChild` class implements an update event for removing a child of a `TMNode`. `TMEventUpdateNewChild` class implements an update for a new child of a `TMNode`. `TMEventUpdateSize` class implements an update size event.

4.1.6 TMExceptionBadTMNodeKind class

`TMExceptionBadTMNodeKind` exception is raised when a `TMComputeSize` or `TMComputeDraw` receive an incompatible kind of `TMNode`.

4.1.7 TMExceptionLeafTMNode class

`TMExceptionLeafTMNode` exception is raised when a `TMNodeModelRoot` received a `TMEventUpdate` event for adding or removing a child to a `TMNode` that is a leaf.

4.1.8 TMExceptionNullParameter class

`TMExceptionNullParameter` exception is raised when null parameter is passed where it should not be.
### 4.1.9 TMExceptionUnknownTMNode class

`TMExceptionUnknownTMNode` exception is raised when a `TMNodeModelRoot` received a `TMEventUpdate` event with a reference to an unknown TMNode.

![Figure 4.9 TMExceptionUnknownTMNode class](image)

### 4.1.10 TMModelNode class

`TMModelNode` interface should be implemented by objects that are model of nodes of the tree that want to be displayed in the Treemap. It’s equivalent of the TreeModel, but for the treemap.

![Figure 4.10 TMModelUpdater, TMModelNode and TMModelUpdaterConcrete classes](image)

### 4.1.11 TMModelUpdater class

`TMModelUpdater` interface represents the object that a `TMModelNode` should call to notify the treemap that something has changed: size, state, number of children etc. A `TMModelNode` should call corresponding update method when something has changed.

### 4.1.12 TMModelUpdaterConcrete class

`TMModelUpdaterConcrete` is responsible for propagating changes in the `TMModelNode` to all views. For the user, the `TMModelUpdaterConcrete` is seen as a `TMModelUpdater` interface. It is the bridge design pattern.
4.1.13 TMNode class

*TMNode* interface should be implemented by the objects that are nodes of the tree and want to be displayed in the Treemap.

---

![Diagram of TMNode classes](image)

Figure 4.11 TMNode, TMNodeAdapter, TMNodeModel, TMNodeEcapsulator, TMNodeModelComposite and TMNodeModelRoot classes
4.1.14 TMNodeAdapter class

*TMNodeAdapter* abstract class encapsulates a *TMNode* for a *TMComputeDraw*. To compute filling and tooltip, a *TMComputeDraw* could need more information of what is stored in the *TMNode*. For example, a filling can be size dependant.

4.1.15 TMNodeEncapsulator class

*TMNodeEncapsulator* implements an encapsulator for user’s nodes when using a *TMModelNode*.

4.1.16 TMNodeModel class

*TMNodeModel* implements encapsulation of a *TMNode* for *TMView*. Its responsibility is to keep the size and the drawing the area and filling of a *TMNode*.

4.1.17 TMNodeModelComposite class

*TMNodeModelComposite* implements the Composite design pattern for *TMNodeModel*. It represents a *TMNodeModel* which is not leaf.

4.1.18 TMNodeModelRoot class

*TMNodeModelRoot* implements the root of the tree of *TMModelNode*.

4.1.19 Thread classes

These classes handle all the applications threads. They are responsible for launching the drawing, redrawing, updating etc., of the treemap.

4.1.20 TMThreadModel Class

*TMThreadModel* abstract class implements a thread of the *TMNodeModelRoot*. Such a thread is launched to update the *TMNodeModel*, or to redraw the drawing buffer of the *TMView*.

```
 treemapSWT::TMThreadModel

 status
 model

 TMThreadModel()
 run()
 guiTask()
 task()
```

Figure 4.12 TMThreadModel class
4.1.21 TMThreadChangeDrawing Class

`TMThreadChangeDrawing` implements a thread that changes the `TMComputeDraw` object used to compute the drawing of `TMNodeModel`.

4.1.22 TMThreadChangeSizing

`TMThreadChangeSizing` implements a thread that changes the `TMComputeSize` object used to compute the size of `TMNodeModel`.

4.1.23 TMThreadLostChild

`TMThreadLostChild` implements a thread that removes a child from `TMNodeModel`.

4.1.24 TMThreadNewChild

`TMThreadNewChild` implements a thread that adds a child to a `TMNodeModel`.

4.1.25 TMThreadTreeBuilder

`TMThreadTreeBuilder` implements a thread that builds the tree of `TMNodeModel` from the tree of `TMNode`. It is the first thread launched by `TMNodeModelRoot` and `TMThreadQueue`. After building the tree it switches the `TMView` to initialized state.

4.1.26 TMThreadLock

`TMThreadLock` implements a really simple mutex.

4.1.27 TMThreadQueue

`TMThreadQueue` implements a que of `TMThreadModel` to be executed in separate threads. With it we can assure that only one update or buffer drawing is occurring at a time.

4.1.28 TMThreadUpdateDraw

`TMThreadUpdateDraw` implements a thread that updates the drawing of `TMNodeModel`.

4.1.29 TMThreadUpdateSize

`TMThreadUpdateSize` implements a thread that update the size of `TMNodeModel`.

4.1.30 TMUpdater Class

`TMUpdater` interface represents the object that `TMNode` should call to notify the treemap that something has changed: size, state, number of children etc. A `TMNode` should call a corresponding method when something has changed.

```java
interface TMUpdater {
    void addChild()
    void removeChild()
    void updateSize()
    void updateState()
}
```

Figure 4.13 TMUpdater Class
4.1.31 TMUpdaterConcrete Class

*TMUpdaterConcrete* is responsible for propagating changes in the *TMNodes* to all views. For this it extends *java.util.Observable* (Observer design pattern). For the user it is seen as a *TMUpdater* interface.

4.1.32 TMView Class

*TMView* implements a view of the treemap. It is responsible for displaying the treemap.

- `addAlgorithm()` – adds a treemap algorithm to this view.
- `setAlgorithm()` – sets the algorithm of the treemap. The name given should be one of the *TMView* constant or name already registered with `addAlgorithm()`.
- `DrawTitles()` – sets the drawing of the nodes titles.
- `getNodeUnderMouse()` - returns the node containing mouse event.
- `zoom()` – zooms the treemap of one level in the direction of the node at the given coordinates.
- `unzoom()` – Zooms out, does nothing if already at the root node.
- `getToolTipText()` – returns the tooltip text.

4.1.33 TreeMap Class

*Treemap* class implements a tree map representation for data. A treemap is built from hierarchical data given as a tree of *TMNode*. So, the first parameter to pass to build a treemap is the *TMNode* which is the root of the to-be-presented tree. The tree to be displayed by the TreeMap should be built before the call to `TreeMap`. 
4.2 Package “project.treemapSWT.graphViewer”
This package is “the application” using the library to view a specific hierarchical data structure. It can also be understood as the frontend of the Treemap Viewer where all the necessary data for building the tree are gathered and passed on to the library. See appendix A.2 for more information.

4.2.1 Demo Class
*Demo* is the class that contains the main method. It all starts here.

4.2.3 Config Class
*Config* is the treemap configuration class. It configures the basic features of the Treemap such as layout algorithm and border size.

4.2.4 Graph Class
*Graph* implements the reading of the directory structures and saving them as a graph into a “*.gml” file.

4.2.5 TMGraphDraw Class
*TMGraphDraw* implements an example of a *TMComputeDraw* for *TMGraphNode*. It uses the name of the file as a tooltip and last modification date for color coding.

4.2.6 TMGraphModelDraw Class
*TMGraphModelDraw* implements an example of *TMComputeDrawAdapter* for *TMGraphModelNode*. It uses nodes properties for color coding and the name of the node for tooltip.

4.2.7 TMGraphNode Class
*TMGraphNode* implements an example of *TMNode* encapsulating graph’s node.

4.3 Implementation Details
This section discusses certain implementation details of the SWT version of the Treemap library.

As stated in the requirements part, the application should be able to read directory structures saved as graphs in “*.gml” files and represent them in the treemap. The code shown in the fig.4.17 does the reading of the “treemap.gml” file and stores the read data in a directed graph. This is currently hard coded, but will be replaced later with a file choose dialog. Then the root node *TMGraphModelNode* is created (line 6). This is necessary for instantiating a treemap processes.

```
1 GML gml = new GML();
2 DirectedGraphInterface graph = new SetBasedDirectedGraph();
3 Reader rd = new FileReader(new File("treemap.gml"));
4 graph = gml.from(rd);
5 DirectedNodeInterface rootNode = FindRootNode.findRootNode(graph);
6 model = new TMGraphModelNode(rootNode);
7 treeMap = new TreeMap(model);
```

Figure 4.17 Code responsible for reading a “*.gml” file and storing the data in a directed graph
The method children() shown in fig.4.18, returns the children of the given node in Enumeration type. If this node is a file it returns an Empty enumeration. But as you can see, this method does the operations on file structures.

```java
public Enumeration children(Object node) {
    Vector children = new Vector();
    if (node instanceof File) {
        File file = (File) node;
        if (file.isDirectory()) {
            String[] tabFichiers = file.list();
            for (int i = 0; i < tabFichiers.length; i++) {
                File fichier =
                    new File(
                        file.getPath() + File.separator + tabFichiers[i]);
                children.add(fichier);
            }
        }
    }
    return children.elements();
}
```

Figure 4.18 Example of the method children()

The method shown in the fig.4.19 is the same method re-implemented to work with DirectedNodes.

```java
public Enumeration<DirectedNodeInterface> children(Object node) {
    Vector<DirectedNodeInterface> children = new Vector<DirectedNodeInterface>();
    if (node instanceof DirectedNodeInterface) {
        DirectedNodeInterface dNode = (DirectedNodeInterface)node;
        EdgeIterator nIterator = null;
        nIterator = (EdgeIterator) dNode.getOutEdges();
        while (nIterator.hasNext()) {
            nIterator.next();
            DirectedEdgeInterface outEdge =
                (DirectedEdgeInterface) nIterator.getEdge();
            children.add(outEdge.getTo());
        }
    }
    return children.elements();
}
```

Figure 4.19 children() method after the reimplementation
In fig.4.20 some basic one to one porting are shown.

<table>
<thead>
<tr>
<th>SWT</th>
<th>AWT/Swing</th>
</tr>
</thead>
<tbody>
<tr>
<td>banner = new Label(this, SWT.NONE);</td>
<td>banner = new JLabel(&quot;Building treemap view...&quot;);</td>
</tr>
<tr>
<td>banner.setText(&quot;Building treemap view...&quot;);</td>
<td></td>
</tr>
<tr>
<td>setSize(new Point(250,60))</td>
<td>setPreferredSize(new Dimension(250,60))</td>
</tr>
<tr>
<td>redraw();</td>
<td>repaint();</td>
</tr>
<tr>
<td>update();</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.20 Basic one to one porting

*SwingUtilities.invokeLater()* method is used when an application thread needs to update GUI. The same thing in SWT does the code on the left hand side of the fig.4.21

<table>
<thead>
<tr>
<th>SWT</th>
<th>AWT/Swing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runnable r = new Runnable() {</td>
<td>SwingUtilities.invokeLater(new NotifyTMSV(increment));</td>
</tr>
<tr>
<td>public void run() {</td>
<td></td>
</tr>
<tr>
<td>new NotifyTMSV(increment);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>if (Display.getCurrent() != null) {</td>
<td></td>
</tr>
<tr>
<td>r.run();</td>
<td></td>
</tr>
<tr>
<td>} else {</td>
<td></td>
</tr>
<tr>
<td>Display.getDefault().asyncExec(r);</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.21 Thread controlling methods

Whenever a painting of some object is involved a paint listener should be added. The code shown in the fig.4.22 is taken from *TMView* class. Before it calls the *paint()* method which does the filling of objects, it first adds a paint listener.

| this.addPaintListener(new PaintListener() { | |
|     public void paintControl(PaintEvent e) { | |
|         GC gc = e.gc;                         | |
|         gc.setClipping(e.x, e.y, e.width, e.height); | |
|         paintMethod.paint(e);                 | |
|     }                                       | |
| });                                        | |

Figure 4.22 PaintListener

In order to get the same functions as in AWT/Swing when it comes to manipulating with mouse, in SWT two mouse listeners were necessary to be added. See fig.4.23.

<table>
<thead>
<tr>
<th>SWT</th>
<th>AWT/Swing</th>
</tr>
</thead>
<tbody>
<tr>
<td>.addMouseListener(new TMAction(this));</td>
<td>addMouseListener(action);</td>
</tr>
<tr>
<td>addMouseTrackListener(new TMAction(this));</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.23 Mouse Listeners
Fig. 4.24 shows the methods that deal with mouse events. The upper part of the figure shows the method implemented in AWT/Swing, the lower part of the figure shows the method implemented in SWT.

```java
public void mouseClicked(MouseEvent e) {
    if (e.isShiftDown()) {
        view.unzoom();
    } else {
        view.zoom(e.getX(), e.getY());
    }
}

public void mouseUp(MouseEvent e) {
    if ((e.stateMask & SWT.SHIFT) == SWT.SHIFT) {
        view.unzoom();
    } else {
        view.zoom(e.getX(), e.getY());
    }
}

public void mouseDown(MouseEvent e) {
}

public void mouseDoubleClick(MouseEvent e) {
}

public void mouseHover(MouseEvent e) {
    view.setToolTipText(view.getToolTipText(e));
    view.getToolTipText();
}

public void mouseEnter(MouseEvent e) {
}

public void mouseExit(MouseEvent e) {
}
```

Figure 4.24 MouseListener methods

In fig.4.25a and fig.4.25b are shown the examples of paint() method implemented in both AWT/Swing (fig.4.25a) and SWT (fig.4.25b). There is no method in SWT that provides the insets of an graphical object. The code in the lower part of the figure is the nearest solution to the code implemented in AWT/Swing. Both methods give almost the same results.

```java
final void paint(Graphics2D g) {
    Insets insets = getInsets();
    root.getRoot().getArea().setBounds(insets.left, insets.top,
    getWidth() - insets.left - insets.right - 1,
    getHeight() - insets.top - insets.bottom - 1);

    root.getLock().lock();
    drawer.draw(g, root.getRoot());
    root.getLock().unlock();
}
```

Figure 4.25a paint() method – original implementation
```java
final void paint(PaintEvent g) {
    Rectangle bounds = getBounds();
    root.getRoot().area.x = bounds.x;
    root.getRoot().area.y = bounds.y - 10;
    root.getRoot().area.width = bounds.width - 10;
    root.getRoot().area.height = bounds.height - 10;

    root.getLock().lock();
    drawer.draw(g, root.getRoot());
    root.getLock().unlock();
}
```

Figure 4.25b paint() method – reimplemented

4.4 Evaluation
Fig.4.26 shows the Treemap application implemented in both SWT and AWT/Swing. The applications look and behave identical. You can also spot the tooltip texts, which display the same information - the name of the node under the mouse. The only differences are the rounded edges of the Group widget displaying TreeMap and the different application icon in the top left corner.

Figure 4.26 The look of the application in both SWT and AWT/Swing
Fig. 4.27 shows both applications after zooming on the same node. The zooming feature has successfully ported as the resulting views are identical.

<table>
<thead>
<tr>
<th>SWT</th>
<th>AWT/Swing</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="SWT Window" /></td>
<td><img src="image2.png" alt="AWT/Swing Window" /></td>
</tr>
</tbody>
</table>

Figure 4.27 Both SWT and AWT/Swing versions of the application after zooming

Fig. 4.28 shows the basic configuration window which is responsible for manipulating the Treemap. “CLASSIC” and “SQUARIFIED” radio buttons change the layout algorithm, while “Set border Size” button sets the border size of the nodes by reading the value entered in the textbox below.

![Configuration Window](image3.png)

Figure 4.28 Configuration window
5. Conclusion and Future work

This chapter gives information about the results of this thesis and about the future work.

5.1 Conclusion

The problem addressed by this thesis was to port of an already implemented treemap library from AWT/Swing to SWT/JFace. Additionally, the library was to be extended so it can read and display graph structures stored in “*.gml” files instead of reading directory structures from the file system. The main reason for porting the application to SWT/JFace is that later it can be integrated as an Eclipse plug-in in the VizzAnalyzer.

We defined five goals to be reached in order to solve the problem. In the following, we discuss how the individual goals were addressed and if the associated criteria were fulfilled.

- The first goal was to write an algorithm, which reads directory structures from file system and converts them to a graph using Grail library. The Grail library has already implemented algorithms for saving the graphs into a “*.gml” file. This goal has been reached since the new class – Graph class, which holds the algorithm for reading and converting the directory structures to a graph has been added to the Treemap library. It can successfully create a tree from a given root folder, containing all files and folders contained by this root element. The details of the class are discussed in Section 4.2.4.

- The second goal was to adjust the Treemap library so it can read graph structures stored in a “*.gml” file and display them into a treemap. The criteria for this goal has been met since the Treemap library was able to display the information stored in a “*.gml” file as same as it displays when it reads directly from the directory structure. This has been discussed in Section 4.3.

- The third goal was to understand the differences and similarities between Java AWT/Swing and SWT/JFace. Gaining this understanding was prerequisite for achieving goals four and five. This goal has been reached since goal four and five has been reached.

- The fourth goal was to understand the implementation of the Treemap library. We divided the classes according to which ones should be ported and which not. Understand the dependencies among them. This was important for the porting phase of the project. It was essential to start from the less important - smaller classes and move on to the bigger ones. Going from outer bounds to the centre of the problem. This was also a research part of the thesis. A UML diagram was used to have a better overview of the whole implementation and ease the understanding of it. This goal has been reached and the essence of the understanding, that is the architecture and design of the TreeMap library, have been documented in Sections 4.1 and 4.

- The fifth and final goal was to port of the implementation from AWT/Swing to SWT/JFace. This was not an easy task because of the different natures of both API-s. Even though most of the porting was 1-to-1 there were still some parts of the application which needed rewriting the code and add new things. The details of this
work have been documented in Section 5 (Implementation). That the goal has been reached all in all has been documented in the evaluation (Section 4.4). Here we showed that the ported application looks and behaves in the same way as the application implemented in AWT/Swing.

Since we reached all stated goals we can conclude that we solved the problem addressed by this thesis.

5.2 Future Work
Adding new features to the application such as representing graph structures in a treemap and porting the application from AWT/Swing to SWT/JFace have successfully met all criteria. This is an important step towards extending the VizzAnalyzer with a more powerful way of selecting the analysis scope of projects and to display hierarchical structures.

Thus, expanding the implementation so it can display containment hierarchy of a software system is one of the main goals for the future. Adding interactive features that besides the containment hierarchy can also show the dependencies among the classes such as inheritance or method calling can ease the understanding of the software structure. For example displaying all the classes that implement certain interface, or all the classes from which a certain method is called, or maybe even combined filtering etc.
6. References


27
Appendices

Appendix A – Package Structure and libraries of Treemap Viewer

A.1 Package project.treemapSWT
This package contains the files necessary for drawing, calculating layout algorithms and
manipulating the treemap. Fig.8.1 shows the list of all the classes that this package consists
of.

<table>
<thead>
<tr>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>project.treemapSWT.TMAction</td>
</tr>
<tr>
<td>project.treemapSWT.TMAlgorithm</td>
</tr>
<tr>
<td>project.treemapSWT.TMAlgorithmClassic</td>
</tr>
<tr>
<td>project.treemapSWT.TMAlgorithmSquarified</td>
</tr>
<tr>
<td>project.treemapSWT.TMComputeDraw</td>
</tr>
<tr>
<td>project.treemapSWT.TMComputeDrawAdapter</td>
</tr>
<tr>
<td>project.treemapSWT.TMComputeSize</td>
</tr>
<tr>
<td>project.treemapSWT.TMComputeSizeAdapter</td>
</tr>
<tr>
<td>project.treemapSWT.TMCushionData</td>
</tr>
<tr>
<td>project.treemapSWT.TMEventUpdate</td>
</tr>
<tr>
<td>project.treemapSWT.TMEventUpdateDraw</td>
</tr>
<tr>
<td>project.treemapSWT.TMEventUpdateLostChild</td>
</tr>
<tr>
<td>project.treemapSWT.TMEventUpdateNewChild</td>
</tr>
<tr>
<td>project.treemapSWT.TMEventUpdateSize</td>
</tr>
<tr>
<td>project.treemapSWT.TMExceptionBadTMNodeKind</td>
</tr>
<tr>
<td>project.treemapSWT.TMExceptionLeafTMNode</td>
</tr>
<tr>
<td>project.treemapSWT.TMExceptionNullParameter</td>
</tr>
<tr>
<td>project.treemapSWT.TMExceptionUnknownTMNode</td>
</tr>
<tr>
<td>project.treemapSWT.TMModelNode</td>
</tr>
<tr>
<td>project.treemapSWT.TMModelUpdater</td>
</tr>
<tr>
<td>project.treemapSWT.TMModelUpdaterConcrete</td>
</tr>
<tr>
<td>project.treemapSWT.TMNode</td>
</tr>
<tr>
<td>project.treemapSWT.TMNodeAdapter</td>
</tr>
<tr>
<td>project.treemapSWT.TMNodeEncapsulator</td>
</tr>
<tr>
<td>project.treemapSWT.TMNodeModel</td>
</tr>
<tr>
<td>project.treemapSWT.TMNodeModelComposite</td>
</tr>
<tr>
<td>project.treemapSWT.TMNodeModelRoot</td>
</tr>
<tr>
<td>project.treemapSWT.TMSDProgressBar</td>
</tr>
<tr>
<td>project.treemapSWT.TMSDProgressSimple</td>
</tr>
<tr>
<td>project.treemapSWT.TMStatusView</td>
</tr>
<tr>
<td>project.treemapSWT.TMThreadChangeDrawing</td>
</tr>
<tr>
<td>project.treemapSWT.TMThreadChangeSizing</td>
</tr>
<tr>
<td>project.treemapSWT.TMThreadLock</td>
</tr>
<tr>
<td>project.treemapSWT.TMThreadLostChild</td>
</tr>
<tr>
<td>project.treemapSWT.TMThreadModel</td>
</tr>
<tr>
<td>project.treemapSWT.TMThreadNewChild</td>
</tr>
<tr>
<td>project.treemapSWT.TMThreadQueue</td>
</tr>
<tr>
<td>project.treemapSWT.TMThreadTreeBuilder</td>
</tr>
<tr>
<td>project.treemapSWT.TMThreadUpdateDraw</td>
</tr>
<tr>
<td>project.treemapSWT.TMThreadUpdateSize</td>
</tr>
<tr>
<td>project.treemapSWT.TMUpdater</td>
</tr>
<tr>
<td>project.treemapSWT.TMView</td>
</tr>
<tr>
<td>project.treemapSWT.TreeMap</td>
</tr>
</tbody>
</table>

Figure A.1 List of classes for treemapSWT package
A.1.1 TMAction class

`TMAction` class manages the action on the treemap. It implements `MouseListener` and `MouseTrackListener` in order to enable mouse actions like clicking and hovering. See fig.8.2.

```java
public void mouseUp(MouseEvent e) {
    if ((e.stateMask & SWT.SHIFT) == SWT.SHIFT) {
        view.unzoom();
    } else {
        view.zoom(e.x, e.y);
    }
}

public void mouseHover(MouseEvent e) {
    view.setToolTipText(view.getToolTipText(e));
    view.getToolTipText();
}

mouseUp() method is called when the mouse button is released.
mouseHover() method is called when cursor stands over the treemap for a short time.

A.1.2 Algorithm classes

`TMAlgorithm` class is an abstract class that represents the algorithm of the TreeMap. It is subclassed by to other classes `TMAlgorithmClassic` and `TMAlgorithmSquarified` which implement a classic and squarified treemap algorithms.

The `draw()` method (fig.8.3) in TMAlgorithm class initializes the drawing process of the treemap.

```java
void draw(PaintEvent g, TMNodeModel root) {
    this.root = root;
    drawNodes(g, root, HORIZONTAL, 1);
}
```

Figure A.3 `draw()` method

A.1.3 Compute classes

`TMComputeDraw` interface should be implemented by every class that implements algorithms for drawing TMNode. The methods to implement are: `getFilling()`, `getToolTip()`, `getTitle()` and `getTitleColor()`. As the filling and tooltip of a node depend on its size, methods get a `TMNodeAdapter` reference and not a TMNode directly. For example, if we want to get the size of a TMNode we call `nodeAdapter.getSize()`, where `nodeAdapter` is a `TMNodeAdapter` reference.

`TMComputeSize` is an interface that should be implemented by every class that implements algorithms for computing the size of TMnode. The computing of the size of TMNode depends of its kind, so `TMComputeSize` should test the kind of TMNodey and throw an exception if there is an incompatibility. `TMComputeSizeAdapter` is the class that implements `TMComputeSize` interface.
A.1.4 EventUpdate classes
EventUpdate classes monitor the changes in TMNode. This TMNode than is encapsulated in the event. TMEventUpdate abstract class implements the command design pattern and should be sub classed to indicate the type of the event to send. These subclasses are TMEventUpdateDraw, TMEventUpdateLostChild, TMEventUpdateNewChild and TMEventUpdateSize.

A.1.5 Exception classes
TMEExceptionBadTMNodeKind exception is raised when TMComputeSize or TMComputeDraw receive and incompatible kind of node.

TMEExceptionLeafTMNode exception is raised in cases of adding or removing a child to a TMNode that is leaf.

TMEExceptionNullParameter is raised when a null parameter is passed where it should not.

TMEExceptionUnknownTMNode is raised when a referencing is done to an unknown node.

A.1.6 Model classes
TMModelNode interface should be implemented by objects that are model of the tree that want to be displayed in the TreeMap.

TMModelUpdater interface represents the object that TMModelNode should call to notify the treemap that something has changed: size, state, number of children, etc. A TMModelNode gets a reference to a TMModelUpdater in the building of the treemap, by the setUpdater() method.

TMModelUpdaterConcrete is responsible for propagating changes in the TMModelNode to all views. For the user it is seen as TMModelUpdater interface, it’s the bridge design pattern.

A.1.7 Node classes
TMNode interface should be implemented by objects that are nodes of the tree and want to be displayed in the treemap. Methods to be implemented are: children() - enumeration type method which returns the children of the current node, isLeaf() – boolean type method which checks if the node is leaf or not and setUpdater() – which is called by the TMUpdater constructor.

TMNodeAdapter abstract class encapsulates a TMNode for a TMComputeDraw. The TMNodeAdapter can give the size and can transmit information between filling and tooltip with getUserData() and setUserData(). The information that could be needed beside that one that is stocked in the TMNode.

TMNodeEncapsulator class implements and encapsulator for user’s node when using a TMModelNode.

TMNodeModel implements encapsulation of a TMNode for the TMView. Its responsibility is to keep the size, drawing area and the filling of TMNode.

TMNodeModelComposite implements the composite design pattern for TMNodeModel. It represents a TMNodeModel which is not a leaf.

TMNodeModelRoot implements the root of the tree of TMNodeModel.
A.1.8 Thread classes
Thread classes are classes that implement threads when particular process or event needs to be done. For example: building a tree, changing the size or the drawing, updating the size or the drawing etc. Threads implemented by TMThreadModel abstract class are launched to update the TMNodeModel (size, drawing, children), or to redraw the drawing buffer if the TMView.

A.1.9 Updater classes
TMUpdater interface represents the object that a TMNode should call to notify the treemap that something has changed: size, state etc. A TMNode should call the corresponding method when something has changed ex: updateSize(), updateState(), addChild(), removeChild(). See fig.8.4.

TMUpdaterConcrete is responsible for propagating changes in the TMNode to all views. For this, it extends java.util.Observable.

public void updateSize(TMNode node) {
    setChanged();
    notifyObservers(new TMEventUpdateSize(node));
}

public void updateState(TMNode node) {
    setChanged();
    notifyObservers(new TMEventUpdateDraw(node));
}

public void addChild(TMNode node, TMNode child) {
    setUpdater(child);
    setChanged();
    notifyObservers(new TMEventUpdateNewChild(node, child));
}

public void removeChild(TMNode node, TMNode child) {
    setChanged();
    notifyObservers(new TMEventUpdateLostChild(node, child));
}

Figure A.4 updateSize(), updateState(), addChild() and removeChild() methods

A.1.10 TMView class
This class implements the view of the TreeMap. It contains different methods for manipulating and drawing the TreeMap, such as: addAlgorithm(), setAlgorithm(), getAlgorithm(), DrawTitles(), getNodeUnderTheMouse(), zoom(), unZoom(), getToolTipText() etc. It also contain some inner classes such as: PaintMethod – abstract class which implements strategy design pattern for the paintComponent() method, EmptyPaintMethod – class which extends PaintMethod and FullPaintMethod – class which also extends PaintMethod class.

A.1.11 TreeMap
This class implements a treemap representation for data. getView() method returns the view of treemap with given renderers. See fig.8.4.
public TMView getView(TMComputeSize cSize, TMComputeDraw cDraw, Composite parent, Display display) {
    if (cSize == null) {
        throw new TMExceptionNullParameter("Impossible to build a treemap" + " view with a null TMComputeSize.");
    }
    if (cDraw == null) {
        throw new TMExceptionNullParameter("Impossible to build a treemap" + " view with a null TMComputeDraw.");
    }
    if (parent == null) {
        throw new TMExceptionNullParameter("Impossible to build a treemap" + " view with a null Shell.");
    }
    return new TMView(updater, cSize, cDraw, parent, display);
}

Figure A.5 getView() method

A.2 Package project.treemapSWT.GraphViewer
This package is the “application” using the library to view a specific hierarchical data. It can also be understood as the frontend of the TreeMap Viewer where all the necessary for building the treemap are gathered and passed on to the treemap library. The fig.8.6 contains the list of classes that this package is consist of.

A.2.1 Config class
This class is simple TreeMap configuration GUI used for configuring the basic features such as layout algorithms and border size. See fig.4.28.

A.2.2 Demo class
This is the class that contains the main method of the TreeMap Viewer. It creates a window used to display the rendered treemap. See fig.4.26 left side.
A.2.3 File properties
This is the class intended to be used on the future work. It expands the file properties for the Grail Library.

A.2.4 Graph class
This class implements the reading of a directory structure and store it as a “*.gml” file.

A.2.5 TMGraphDraw class
This class implements an example of a TMComputeDraw for TMGraphNode. It uses the name of the file as a tooltip and last modification date for colour coding. It contains method like getFilling(), getTooltip(), getTitleColor() and getTitle().

A.2.6 TMGraphModelDraw
This class implements an example of TMComputeDrawAdapter for TMGraphModelNode. It uses nodes properties for colour coding and the name of the node for tooltip. It contains methods such as getFillingOfObject(), getToolTipOfObject(), getTitleOfObject and getColorTitleOfObject().

A.2.7 TMGraphModelNode
File encapsulator.

A.2.8 TMGraphModelSize
This class implements an example of a TMComputeSizeAdapter for a TMFileModelNode. The method getSizeOfObject() is used for returning the size of the node. See fig.8.7.

```java
public float getSizeOfObject(Object node) {
    if (node instanceof DirectedNodeInterface) {
        DirectedNodeInterface dNode = (DirectedNodeInterface) node;
        return dNode.getFile().length;
    }
    return 0.0f;
}
```

Figure A.7 getSizeOfObject() method

A.2.9 TMGraphNode
This class implements an example of TMNode encapsulating a file. It also contains some methods used for obtaining basic attributes of a node (files) such as getFullName() – returns the full name of the file, getName() – returns the name of the file, getSize() – if the node is a file returns the size of the file otherwise it returns 0, getDate() – returns the last modification date, children() – returns the children of a node in an Enumeration, and isLeaf() – returns true if the node is not a directory.

A.2.10 TMGraphSize
This class implements an example of a TMComputeSize for a TMFileNode.

A.2.11 TMGraphSizeDate
Same as TMGraphSize except it uses the date of a file as a size