Juha Hollanti

Representing Knowledge

Thesis
CENTRAL OSTROBOTHNIA UNIVERSITY OF APPLIED SCIENCES
Degree Programme for Information Technology
November 2010

Thesis Abstract

Department Technology and Business, Kokkola	**Date** 22 Nov 2010	**Author** Juha Hollanti
Degree programme Degree Programme for Information Technology		
Name of thesis Representing Knowledge		
Instructor Szewczyk Grzegorz		**Pages** 88 + appendices
Supervisor Szewczyk Grzegorz		

This thesis work was about creating a commissioned work for a company called ALMA Consulting Ltd. The commission work was about creating a visual representation for the system that ALMA is building. The ALMA system is an information and knowledge management system for factories and production facilities.

This work is also about investigating the user interface differences between two technologies – the Java platform and Qt UI library. This comparison proved to be difficult to make and the results found are somewhat ambiguous. The comparison fell short because of a lack for a proper graphing library for Qt.

The goal of this work was to build an application and to build this application a software methodology titled Unified Process was used. This report extensively tries to describe the Unified Process and how a person might employ it in use in Software projects. This report was built and documented in an Agile project manner, which makes it a little bit different from a normal approach to writing a project document.

Keywords:
ALMA, Software Engineering, Graph Visualization, Visualization of Objects, Java, Qt,

Abstrakti

Osasto Tekniikka ja Liiketalous, Kokkola	**Date** 22.11 2010	**Tekijä** Juha Hollanti
Koulutusohjelma Tietotekniikan koulutusohjelma		
Opinnäytetyön nimi Representing Knowledge		
Ohjaaja Szewczyk Grzegorz		**Sivuja** 88 + liitteet
Valvoja Szewczyk Grzegorz		
Tämän työn tarkoitus oli luoda sovellus yritykselle nimeltään ALMA Consulting Oy. Sovelluksen tarkoitus on visuaalisesti tuoda esille yrityksen luoman sovelluksen, ALMA järjestelmän, rakenne. ALMA on tietämyshallintajärjestelmä teknisen tiedon ja tapahtumien elinkaarenaikaiseen hallintaan. Työn tavoite saavutettiin onnistuneesti. Tämän työn oli myös tarkoitus verrata toisiinsa Javan ja Qt:n tarjoamia käyttöliittymäsuunnittelua. Tässä työssä käytettiin sovelluskehitysmenetelmää nimeltään Unified Process. Tämän työn tarkoitus oli myös kuvata, miten tätä menetelmää käyttäen luodaan sovelluksia ja tama työ dokumentoitiin Agile kehitysmenetelmän tavoin. Tämän takia raportin rakenne poikkeaa jokseenkin perinteisestä raportin rakenteesta.		
Hakusanat: Graph		

TERMS

Agile	A broad term to house a number of modern software engineering methodologies. The most common Agile methodologies are Scrum and UP with its different variations such as RUP.
ALMA	Automation Lifecycle Management. Because ALMA is both an acronym and a company, it creates a source of ambiguity. In this work ALMA is never used as an acronym. When speaking of ALMA I'm always referring to the company, not the acronym.
ALMA Consulting Ltd.	The company, which commissioned the work discussed in this report.
AWT	Abstract Window Toolkit. The original windowing, graphics and user-interface widget toolkit for Java.
BFS	Breadth First Search. A search used with graph structures. Contrast with DFS.
BGL	Boost Graph Library. C++ library for analyzing graphs.
Boost	A set of various C++ libraries.
CVS	Concurrent Versioning System. Version control tool.
Design Pattern	A general, i.e. reusable, solution to a software engineering problem.
DFS	Depth First Search. A search used with graph structures. Contrast with BFS.
Edge	Graph element. Connects two nodes to each other.
Graph (Graph Theory)	The word *graph* is used in different context where it exhibits different meaning. I want to set apart the graphs discussed in this work from the most popular notion of a graph, i.e. a graph of a function. The graphs discussed in this work are graphs of Graph Theory, not graphs of a function.
Graph (Graph of a Function)	Graph of a function plots how the output of a

	function changes in respect to the input it receives.
GraphML	Graph Markup Language. A standard markup for graphs.
HWA	Heat, Water, Air-condition.
IDE	Integrated Development Environment. Tool of a developer, which includes compilers, builders, etc. in one concise package.
Interface (Programming)	A programming language construct, which allows the design of robust and scalable systems.
Interface (API)	Application Programming Interface (API) describes how to use an external (3rd party) application via an interface designed specifically for this purpose.
Interface (Between Technologies)	For example, an interface between Java and C++, which is one aspect of this whole work.
JNI	Java Native Interface. A technology to interface Java technology with operating system native technology.
JVM	Java Virtual Machine. JVM executes Java byte code.
MVC	Model-View-Controller. A Design Pattern used in most systems, which offer user interface elements.
Node	Graph element. A nexus point to which edges connect to.
OOP	Object Oriented Programming. OOP is the mainstream solution for managing large software projects.
OOA/D	Object Oriented Analysis and Design. The concept of modeling and designing software systems.
OGDF	Open Graph Drawing Framework. A C++ library for graph drawing.
Qt	Qt application framework. A cross platform library to create native software. Primarily advertised for its effectiveness in UI development.

Swing	Swing supersedes AWT by providing a richer set of UI widgets.
TDD	Test Driven Development. Software engineering method.
UI	User Interface.
UML	Unified Modelling Language. General purpose modeling language for software engineering.
UP	Unified Process. An iterative software development methodology.
Set (Mathematics)	Unordered set of arbitrary items.
VCS	Version Control System.
Vertex	Synonym for node.
Waterfall	Software development methodology, which emphasizes planning and documentation.

TABLE OF CONTENTS

1 INTRODUCTION

Information and structure is best described using images and symbols. From a human point of view a visual representation can be consumed much more quickly and much more efficiently. This thesis works in this context—in the context of visualizing data, information and knowledge. This thesis work was commissioned by ALMA Consulting Ltd.

This work also touches upon the issue of user interface (UI) development—especially, maturity of UI development between different platforms. Two technologies are contrasted between each other, the Java platform and Ot application development framework. Drawing straight on conclusions between Java and Qt can be deemed as a naive task since the technologies vary and work in different areas—Java is a full-blown platform whilst Qt is "just" a UI library. However, Qt offers functionality past UI and what Qt lacks can be taken care of with C++ since Qt works on top of C++.

This work is primarily about software engineering. There's of course theory about networking and graphs but all those aspects are rather well taken care of by ready-made libraries used in this work—I won't go into great detail about graphing. So, when reading this take into consideration that it mainly describes the software engineering process I followed when creating new software. This software engineering process is thoroughly discussed in chapter 4.7 .

All the images and pictures in this work are title as graphs. So you'll see images and pictures of class diagrams, sequence diagrams, etc., titled as graph 1, graph 2, etc. These pictures and images, though titled as graphs, are not to be confused with the graphs discussed in this thesis work. The graphs discussed in this work are data structures. A more detailed description can be found from chapter 4.1 .

2 GOALS AND PROBLEM SETTING

The primary goal of this work is, of course, to provide a working solution for the commissioner of the work. This is nothing too amazing in itself; it's basically to employ the use of a Java graphing library to represent information and knowledge in a clear and concise way.

Secondary goal is to contrast Qt with Java, or more precisely, Qt in combination with C++. The secondary goal is a "nice-to-have" and it's purely an additional goal, which is greatly shadowed by the primary goal.

Also posing a problem is to develop a Java and a Qt application simultaneously so that they could be effectively compared. The two projects overlap by some but there are also aspects, which vary greatly between the two.

2.1 ALMA Consulting Ltd.

The company, which commissioned the work, ALMA Consulting Ltd, is a knowledge-based company founded in 1986; currently located in Kokkola. To describe the company, the following excerpt is from the ALMA website:

> ALMA® is a totally integrated engineering and knowledge management system for creation and life-cycle handling of Logical Plant Model, technical data and documentation, production line efficiency and maintenance management. (ALMA Consulting Ltd. 2010)

2.2 Goals

The goal of this work is to produce sensible means to portray the ALMA information system in a graphical setting. This means providing clear and concise visual representation of information about different elements in the information

system and how they relate to each other. The ALMA knowledge management system is an object-based system where objects can relate to each other by hierarchy and links. For a naive comparison, it could be contrasted with the structure of the World Wide Web.

2.3 Context

The ALMA system is an information system for plants and production facilities in general. Its main purpose is to document in some way or the other all that happens in a production system. ALMA offers products for Process Automation, Field Engineering, Electrical Engineering, Mechanical Engineering, Construction, Maintenance, HWA and Documentation. All the different products (or modules) work on top of a core application, which is responsible for composing tailored solutions for different needs. ALMA uses server-client based architecture. Both the server and the client are realized using Java technology. For the client, ALMA offers a desktop application for all systems that support Java, as well as a web interface for browsers. The ALMA system is integrated to 3rd party systems where necessary.

All this varying information and functionality should be able to be expressed by visual means—i.e. by drawing a graph—which shows how different parts interact with each other.

More detailed description of ALMA system is considered sensitive material and it shall not be discussed in this work.

2.4 Limitations of the work

This work aims to describe the structure of the ALMA system and how to implement a graphical graph representation on top of it. This work does not try to describe how exactly the ALMA system might be used—this would be the subject of a completely different thesis work.

2.5 Structure of this document

This document first describes the initial settings and requirements of the work. After that all the theoretical aspects are accounted for and last the work done is documented.

This work was done using Agile software methodologies, which means that contrary to traditional software development, no exhaustive plans were made before the work was started. Every effort has been made to adhere to Agile software development methods and Agile modelling (see chapter 4.7) when creating the documentation. (Larman 2005, 17-22)

All major, and some of the minor use cases—which are not that numerous—are documented and in this fashion the future development of the system is also going to be, at least partially, documented. These requirements have been mainly gathered by conducting meetings in the workplace where people suggested functionalities that would be useful for the end users. On top of these "pre-made" requirements actual customers have suggested functionalities that they would need to have in the product so that it would better produce value for them. I also insisted on a few of my own functionalities.

3 INITIAL SETTINGS AND OVERVIEW OF THE ACHIEVABLE RESULT

The primary thesis of this work is to provide a working solution to visually present the ALMA system. There are varying aspects to this, which are included next.

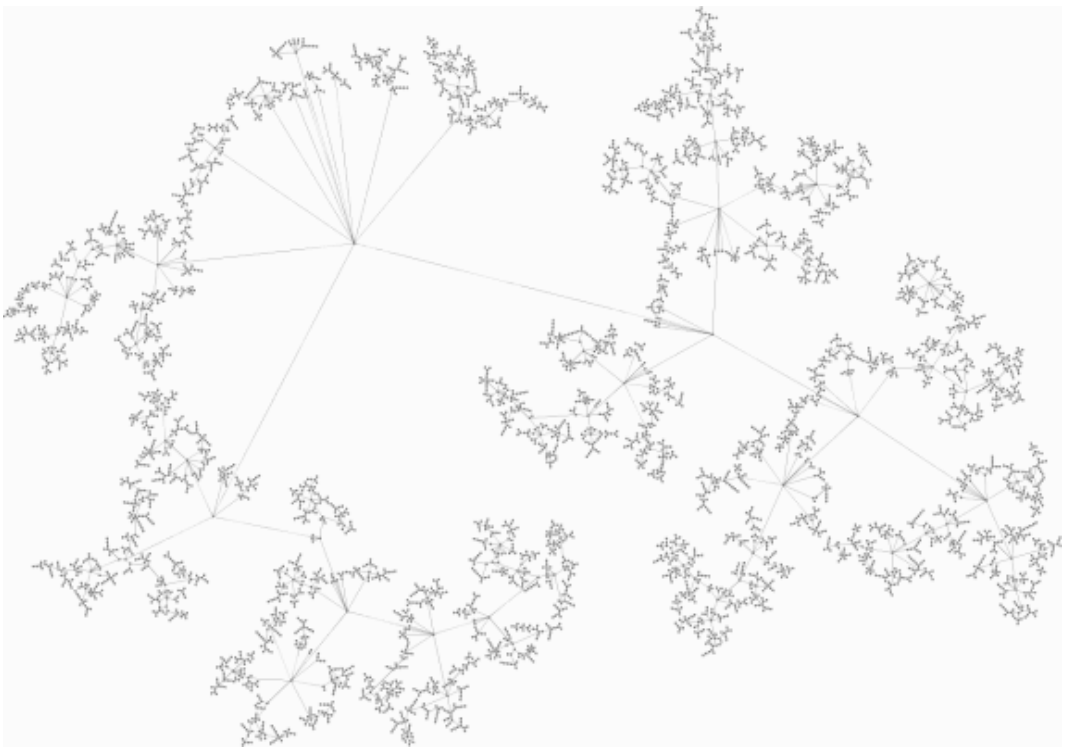
3.1 Hierarchical views

To get the best picture of how a production facility is structured a hierarchical view is the best choice. This is also a good way to introduce the layout of the production facility to new, and maybe even to old, personnel. It provides a clear and concise way to understand the lay of the land.

3.2 Network views

A network view is basically the same as a hierarchic view but it also supports inspection of the link structure. The need for network views is to quickly realize the structure of information (or data) flow in the system. This could, for instance, be used to quickly realize any possible weaknesses in the information flow.

Picture below depicts an example of a network view of an arbitrary structure. It might be difficult to interpret this picture. The picture consists of graph elements, nodes and edges. A node is a dot on the picture. Each node is connected to an arbitrary amount of other nodes by a line leading from one node to the other. This connecting line is called an edge. This essentially forms a graph. In this case the graph is connected because you can start from any given node in the picture and by following edges you can reach all nodes in the graph.



Graph 1: Example of a networked graph structure (yWorks 2010)

As an example, imagine a production facility that uses automated sensor to signal the start of a production cycle. This could be something like a proper amount of weight of substance in a conveyor line before production can be started. Now imagine that this sensor is not connected to the system or perhaps it's connected but in an incorrect way. To spot these kinds of deficiencies is critical because a production facility cannot tolerate constant pauses, which cause loss of resources and time.

3.3 Schematics

Schematics are used to display connections, e.g. from a sensor to chip. The world of schematics is also a very broad and deep one. In this work a basic implementation for schematics is included, but it would be too much to work out an exhaustive implementation.

3.4 Grouping and abstractions

An important aspect of information visualisation is to group and abstract certain things. For instance, same kinds of things should be grouped and some things should be abstracted away. An example of an abstraction would be to represent a cable instead of all the wires it holds within it.

Creating groups and abstractions in a graph will quickly add to the complexity of the whole application since in a graph, all functionality either indirectly or directly affects other functionality. E.g. grouping might affect abstractions and vice-versa, because elements of a graph might belong to both sets. Grouping and abstractions directly affect the basic elements of a graph. This creates an intertwined functionality mess, which is difficult to implement and maintain (coding wise).

4 THEORY

Graph theory contains within it all the theoretical aspects of this work. To begin with, I'm going to describe the basic nature of graphs and trees (since trees are perhaps the most common type of graph) and how they're used utilized in the work. I'm also going to describe other data structures used in the work and the most common/important algorithms used.

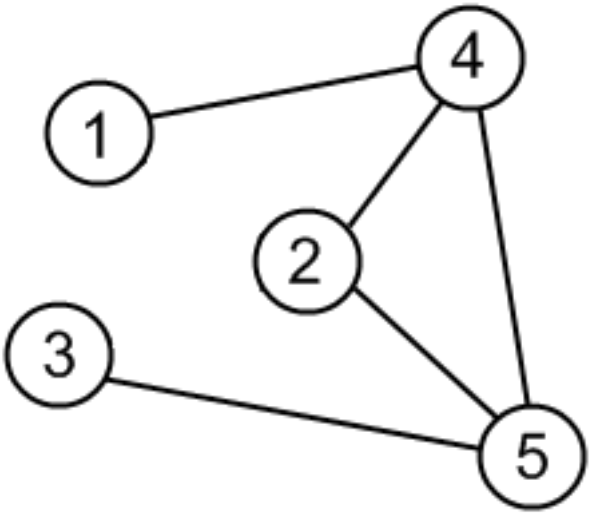
I'll also describe the software engineering methods used and I'll provide quick explanations for the Design Patterns used in this work because in chapter 47 I will refer to these structures rather often.

I'll also describe a few algorithms. Though this is largely unnecessary, it does paint a picture of how one works with graphs.

4.1 Graphs

A graph is a data structure in computer science used to model the relationships between objects. These objects can be whatever the problem requires them to be. A graph consists of nodes and edges. A node is a nexus point to which edges can connect to. Two edges cannot connect to each other. Thus graphs realize network structures, which again can be used to model things like the World Wide Web. (Knuth 1997, 363-372)

The following picture depicts a simple graph.



Graph 2: Simple graph

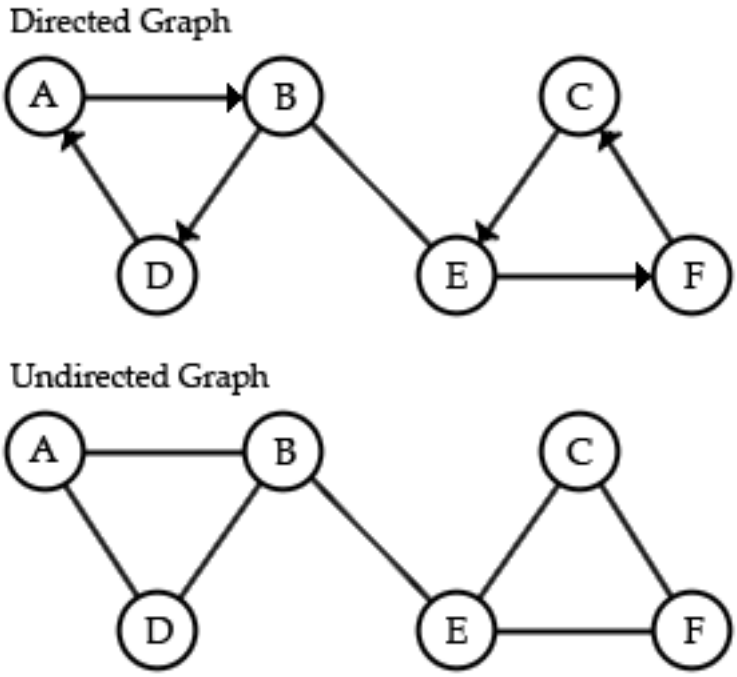
There exist various definitions and various different semantics for graphs with differing natures. Concerning this work, it's not necessary to go over different definitions. Suffice to say that a graph can take on various characteristics, which quickly turn it from one thing to another.

As a single example, consider the definition for a tree. A tree is said to be a connected, undirected, acyclic graph (Black, tree 2008). It is in this fashion that different kinds of graphs are labeled. *Connected* means that if you start from any node in the graph, you can reach any other node by traveling via edges (Black, connected graph 2004). *Acyclic* means that there are no cycles formed in the graph structure (Black, acyclic graph 2004). Adding a single edge to the tree found from Graph 4: Simple tree structure would render the graph cyclic (consider Graph 2: Simple graph) and it could no longer be seen as a tree in the meaning of the word. Removing an edge renders the tree as a forest (more on forests shortly). This means that a tree always has an edge count of $n - 1$, where n is the amount of nodes.

If the graph was not *connected* but was *acyclic*, it would mean that the graph would consist of a number of trees. This would essentially render the graph to bear the

label forest (Black, forest 2004). So, a forest is a group of trees—a quite fun, and fitting, multilevel analogy. A forest could then be defined as having an edge count of $(n-1) - (t-1)$, where n is the amount of nodes in the graph while t is the amount of trees in the graph.

In addition to the definitions found from this simple definition of a tree, a graph can be considered *directed* or *undirected*. This simply states whether the edges of the graph are directed or not—that is, an edge can only be traversed one way so that if an edge exists between nodes A and B and the edge is directed, meaning that it travels from node A to node B, you can only traverse the edge from node A to node B (Black, directed graph 2008). Consider the following graph:



Graph 3: Directed and Undirected Graphs side by side

This gives new meaning to *cyclic* and *acyclic*, since it's not a given that a *cyclic* graph stays cyclic if you superimpose the meaning of directed edges upon it.

There also exist a variety of terms, which further specify a given structure. A tree could, for instance, bear an add-on label of *binary*, making it a *binary* tree. A *binary*

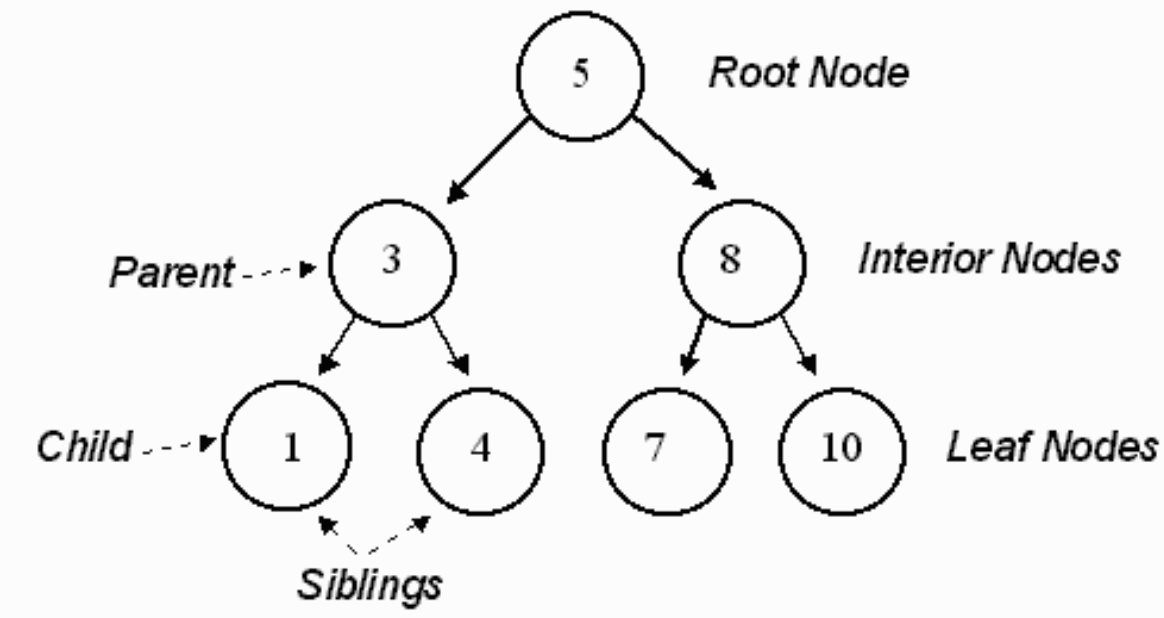
tree could, once again, be furthermore labeled e.g. *red-black* tree or it could be generalized into a *B-tree* and/or a number of other labels could be assigned to it (Black, red-black tree 2011). Concerning this work, it doesn't really matter what all of these labels mean, suffice to say that there are very many of them.

Concerning data structures, graphs are usually implemented with maps, where graph elements and connections between the elements are stored in maps. A graph is not an overly complex structure but surprisingly enough, most problems can be modelled into graph problems. (Yegge 2008)

4.2 Trees

Trees are the most important nonlinear structure in computer algorithms. Nonlinear means that you can't have, except on special cases, linear trees—that would effectively render them as lists. Like the graph, a tree is a data structure consisting of nodes and edges—more precisely, tree is a specialization of a graph. Trees differ from graphs in the sense that a tree cannot have within it loops. (Knuth 1997, 308)

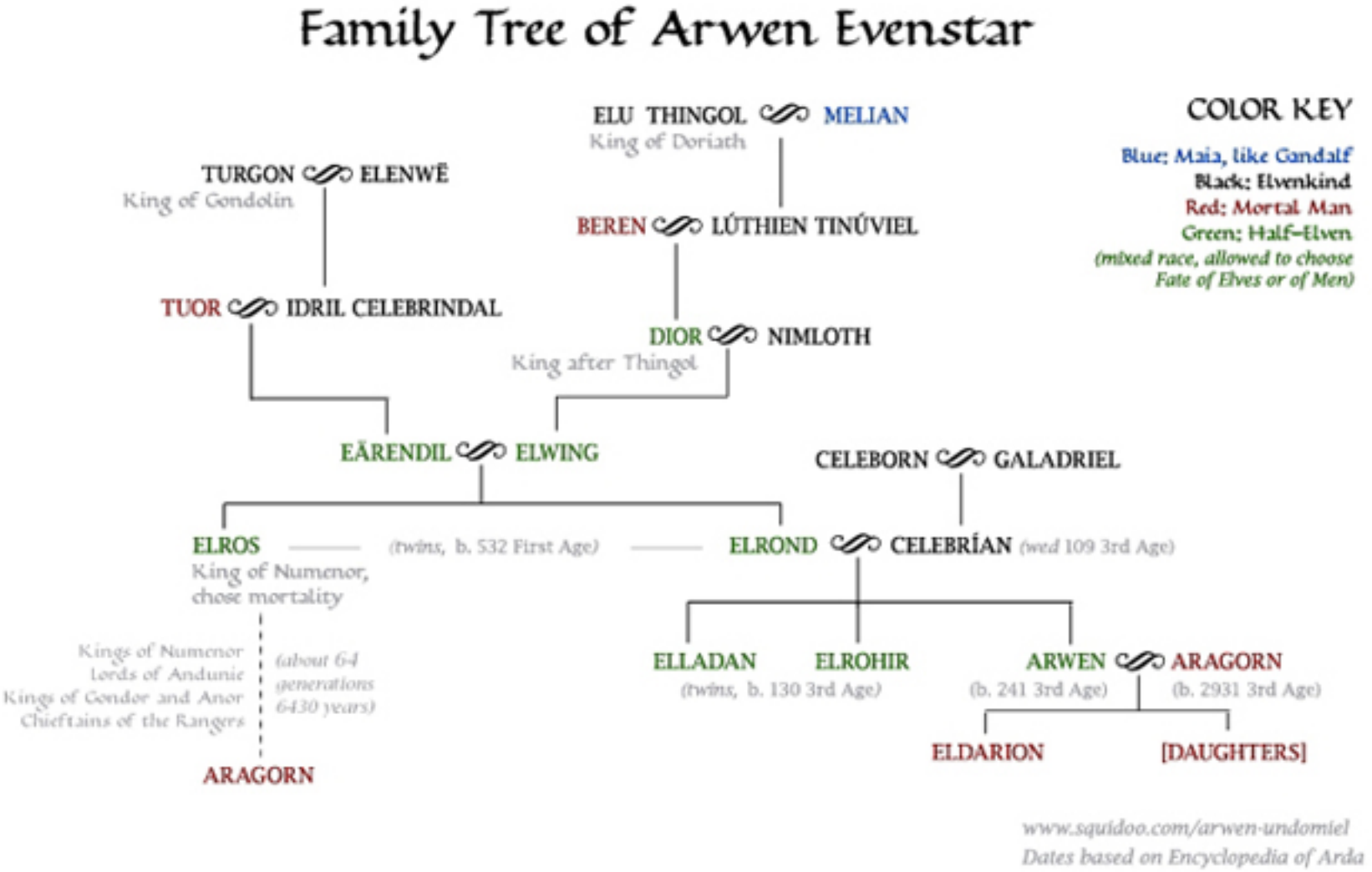
The following picture describes a simple tree. The numbers inside the nodes is not important in this context.



Graph 4: Simple tree structure (Holowczak 2007)

As can be seen from the picture, a tree—any graph for that matter—has the following elements in it: root node, leaf nodes, parents, children and siblings. The definitions parents, children and siblings are of course context sensitive—different nodes have different relationships, i.e. different parents and/or children and/or siblings. Two nodes cannot have same relations; otherwise they would essentially be the same element.

Trees are especially useful when representing hierarchies. Consider Graph 5: Family tree of Arwen Evenstar of a simple family tree. The hierarchy is easy to spot and it's easy to see how everybody relates to each other.



Graph 5: Family tree of Arwen Evenstar (Squidoo 2011)

4.3 Maps, Lists and Sets

Graphs and trees are the primary ways to visualize an information system like the ALMA system is. However to support these data structures a variety of other data structures are utilized.

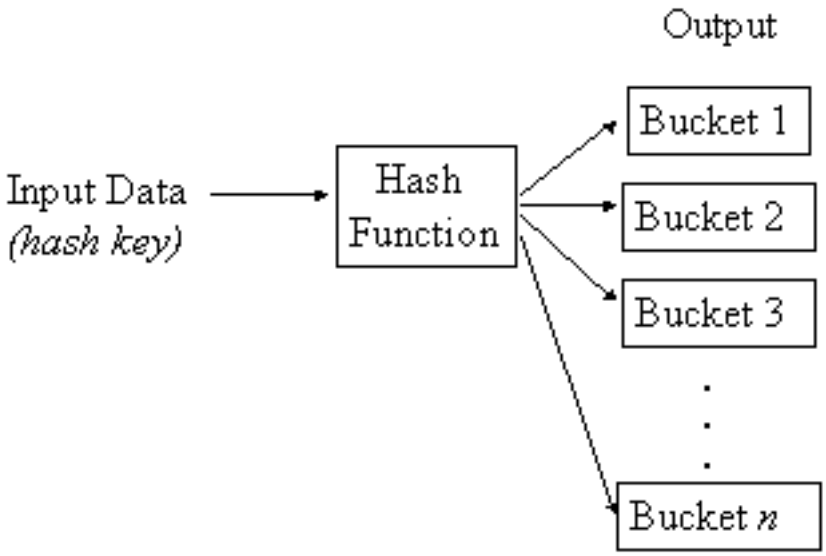
A map is a data structure, which associates a value to a key. The key and the value can be anything that will fit the bill. (Vesterholm and Kyppö 2006, 290, 300, Niemeyer and Knudsen 2005, 361-363)

A list is an ordered (i.e. sequential) set of items. It can hold duplicate values in it and it is usually used to hold an array of similar kinds of items within it. The list is most likely the most used data structure in applications of any kind. (Vesterholm and Kyppö 2006, 290-292)

A set on the other hand does not allow duplicate values in it nor does it keep an order of any kind in it. A set (at least concerning Java) consists of homogeneous types of objects. (Vesterholm and Kyppö 2006, 290-292)

4.4 Miscellaneous

When dealing with maps, quite frequently the maps used are hashed maps. A hashed map uses a hash value of an object instead of the object itself the key. A hash value is a simple value, such as an integer, given to a complex structure, such as an object. This way it is easier and faster to work with complex structures. Invariably, the maps used in this work are going to be hashed maps because they offer constant time for adding, removing and retrieving an element. The one drawback that hashed maps introduce is that if you're extremely unlucky you might get collisions when converting complex structures into simple hashed values. This does not happen often and usually it can be guaranteed not to happen.



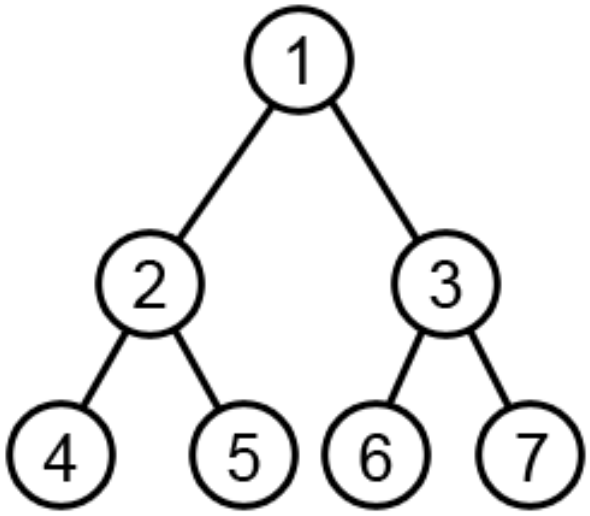
Graph 6: Hashing example

In Java, a map needs to be typed, i.e. you need to define what kinds of elements you want to store in it. This is perfectly fine. Java does not, however, support the use of primitive types as map types. This means that instead of using a number, Boolean, etc. as the type, you need to use the primitive types object type. (Niemeyer and Knudsen 2005) The object types work exactly the same as primitive types but they consume more memory and are slower to work with. To come around these limitations an external library is used, when applicable, called fastutil, which offers maps (and other data structures too), which can be typed with primitive types. The functionality remains exactly the same; the performance is a little bit better, especially when it comes to preserving memory. The difference is small but with large structures it helps a lot.

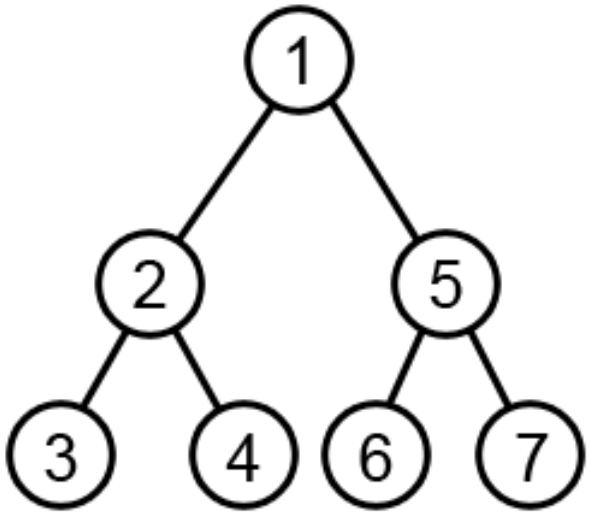
Traversing a graph can be done in a number of ways but they usually fall into two categories: breadth first and depth first. Most often, when a graph is traversed it is done in order to find an item. Search-wise, corresponding with breadth first and depth first are Breadth First Search (BFS) and Depth First Search (DFS). The names rather successfully describe how the searches work. BFS searches for an item, trying to find it first as near as possible, DFS quickly starts to look for the item further away. BFS is especially useful if you want to limit the number of recursion—i.e. the maximum depth of how far away should the item be looked for

(Knuth 1997, 351). BFS is also very useful if you want to search for the nearest matching element (Knuth 1997, 351). The implementation of a DFS is a bit more straightforward than that of BFS because a DFS can be implemented by using recursion alone. (Knuth 1997, 437)

Consider the following graphs. The number inside a node tells the order in which it is processed; contrast the difference between BFS and DFS.

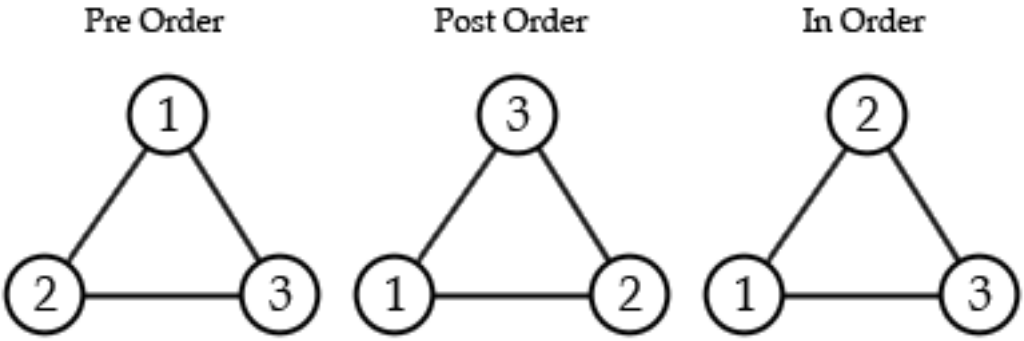


Graph 7: BFS example



Graph 8: DFS example

Other ways of traversing graphs are Pre Order, Post Order and In Order. These methods may very well be more common than BFS or DFS. Consider the following graph:



Graph 9: Pre Order, Post Order and In Order ways to traverse a graph

Preorder traversal first considers the node itself, then its left descendants and finally its right descendants (Black, postorder traversal 2008). Post Order traversal first considers a nodes left descendant, then its right descendant and finally the

node itself (Black, postorder traversal 2008). In Order traversal first considers the nodes descendant on its left, then the node itself and finally the descendant on its right (Black, in-order traversal 2008).

Often when traversing a graph, a technique called coloring is used. Coloring is adding information to a node to ensure, for instance, that a node is processed only once, or some other predefined number of times. This is important when loops arise in a graph structure—you need to somehow be able to tell what nodes you have gone through already, otherwise you would spend the rest of eternity traversing the graph.

Sometimes when traversing a graph, the structure of the graph is first flattened or linearized before it is processed. This means that the relationship information is abstracted away from the graph—the nodes are stored as an ordered set (i.e. a list) while the edges are ignored altogether. A graph can be flattened using a number of strategies. E.g., left-to-right flattened list of the graph found from Graph 8: DFS example gives a list of: 3, 2, 4, 1, 6, 5, and 7. Top-to-bottom would yield: 1, 2, 5, 3, 4, 6 and 7. There are numerous ways to flattening a graph.

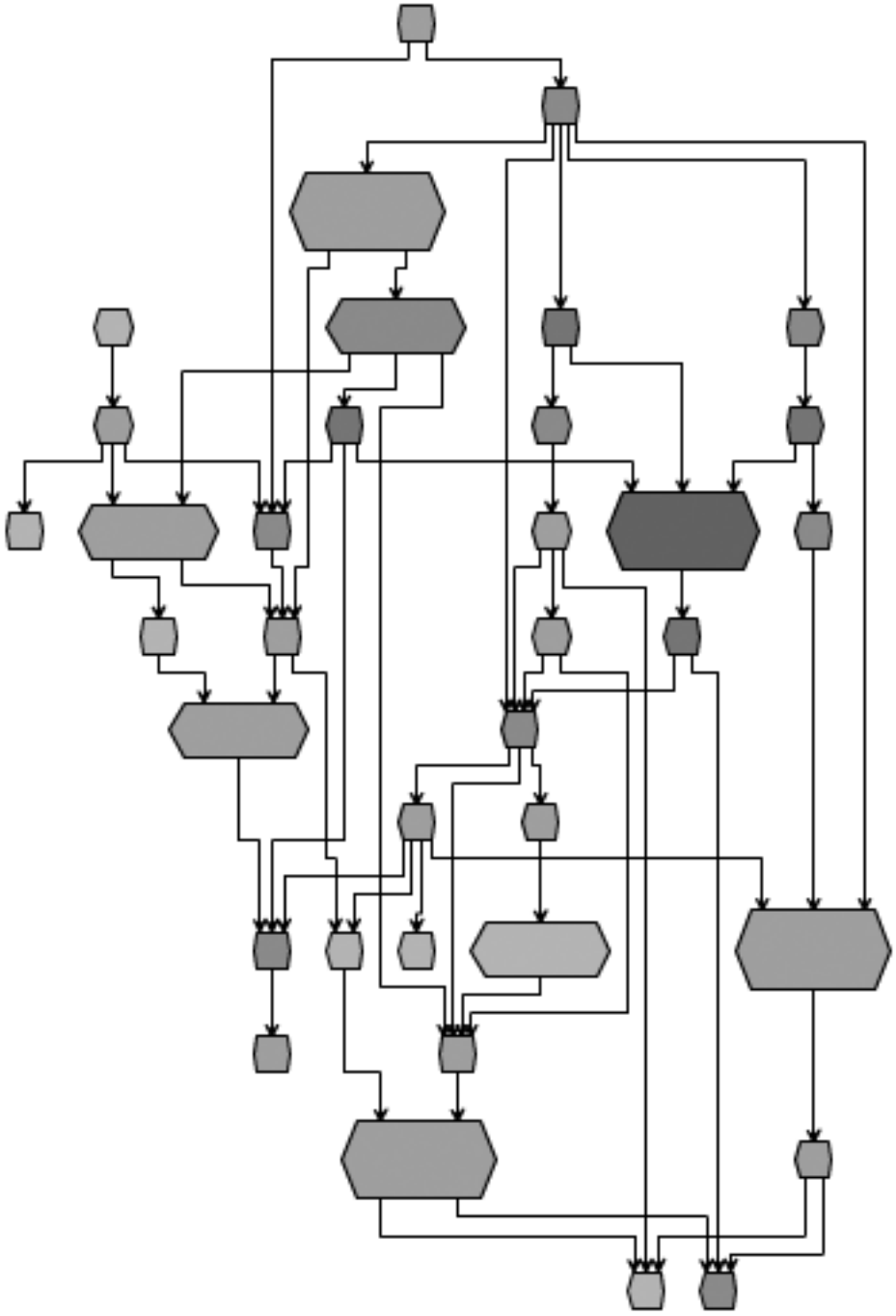
4.5 Layouting Graphs

An important aspect of creating graphs and trees is to visually layout them to the screen. This is primarily where the yFiles Java library comes in. yFiles provides a number of layout algorithms. The primary layout methods used in this body of work are: Hierarchic, Organic, Balloon ... etc. The names of the layout algorithms quite successfully describe how they function.

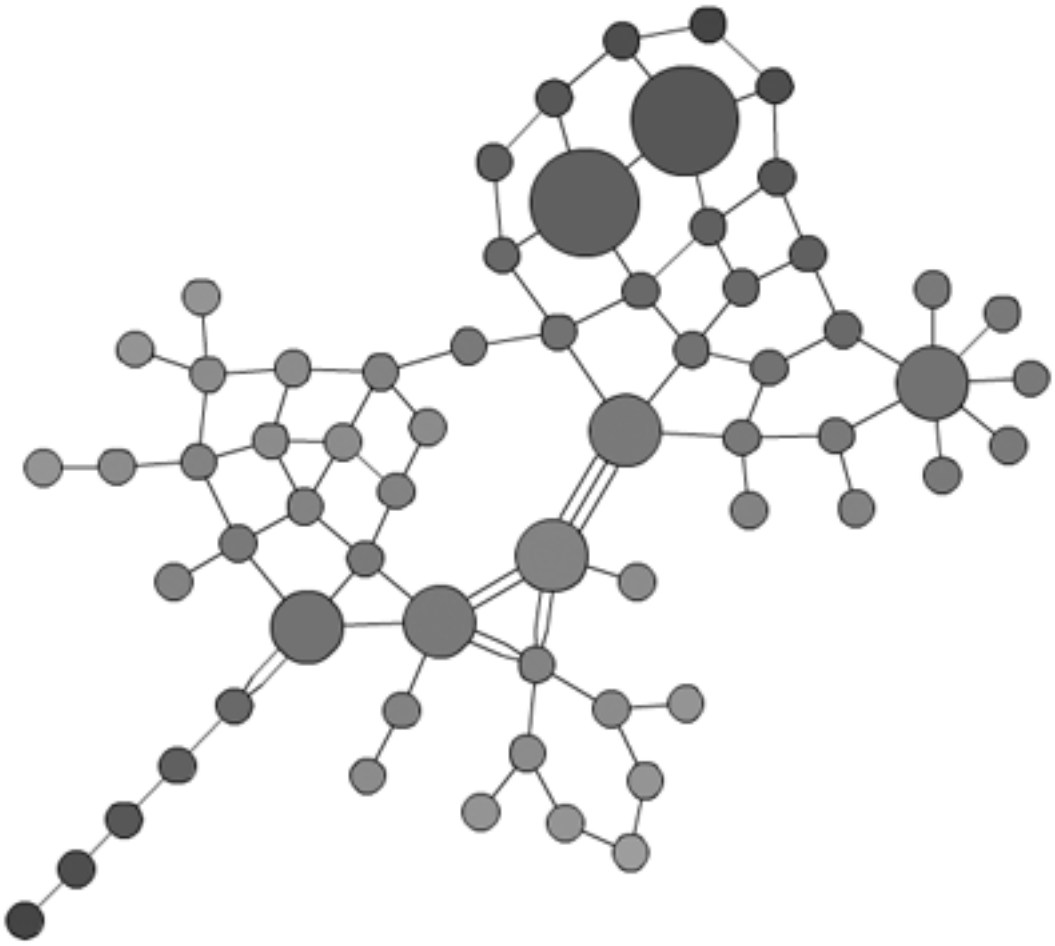
Layouting is an important aspect of this work because with different structures different layout algorithms provide views, which are easier for human beings to interpret visually. Hierarchical views, for instance, are ideal when the hierarchical nature of a tree is presented. If included in that same tree is the link structure, a different layout algorithm might portray information in a better way. There is no

simple way to say where to use a given layout algorithm, which only stresses the importance of allowing the user to specify it by themselves.

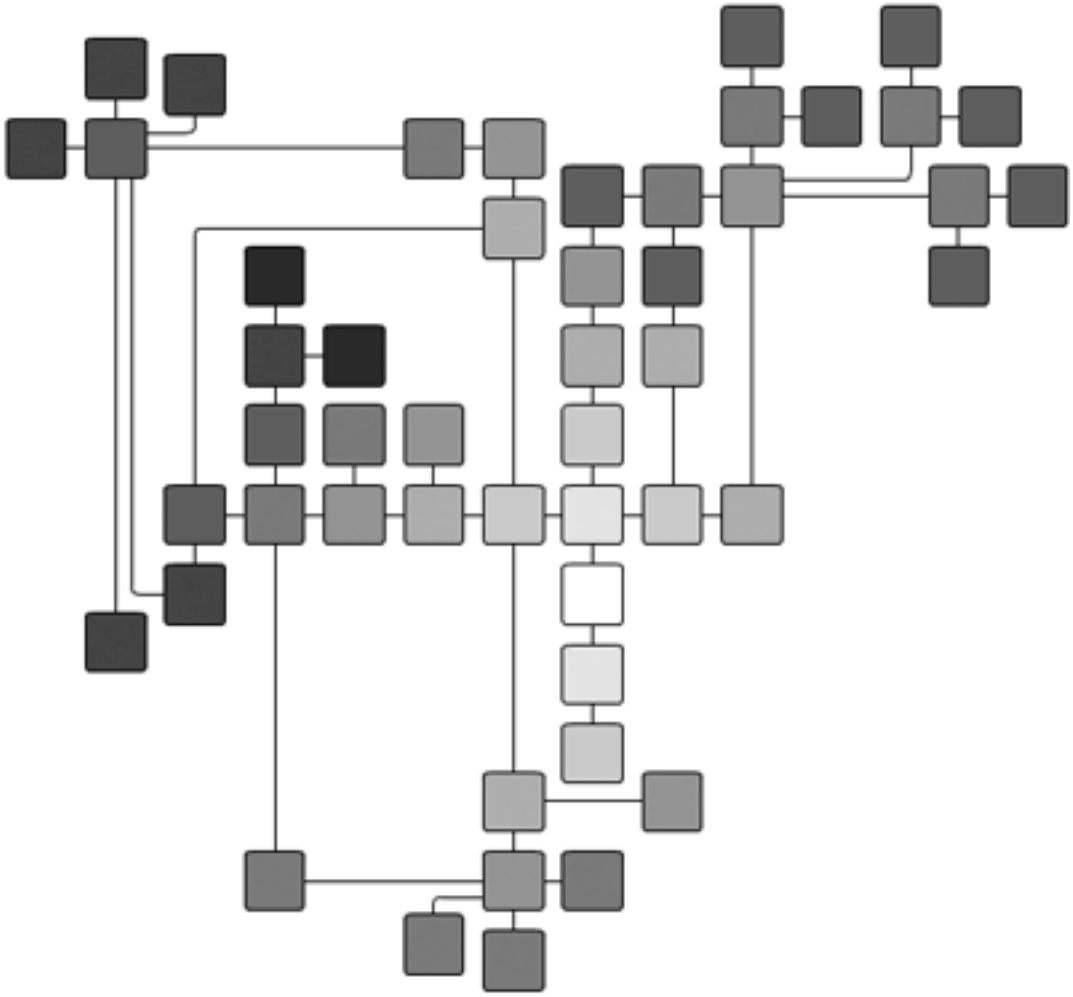
Below are pictures of graphs layouted with various layout algorithms. Note the picture rendered with Circular layouter—a graph can quickly grow to be hard to understand. The nodes and edges are of varying shapes and sizes in the pictures. This goes only to show that it's possible to create nodes and edges with varying characteristics.



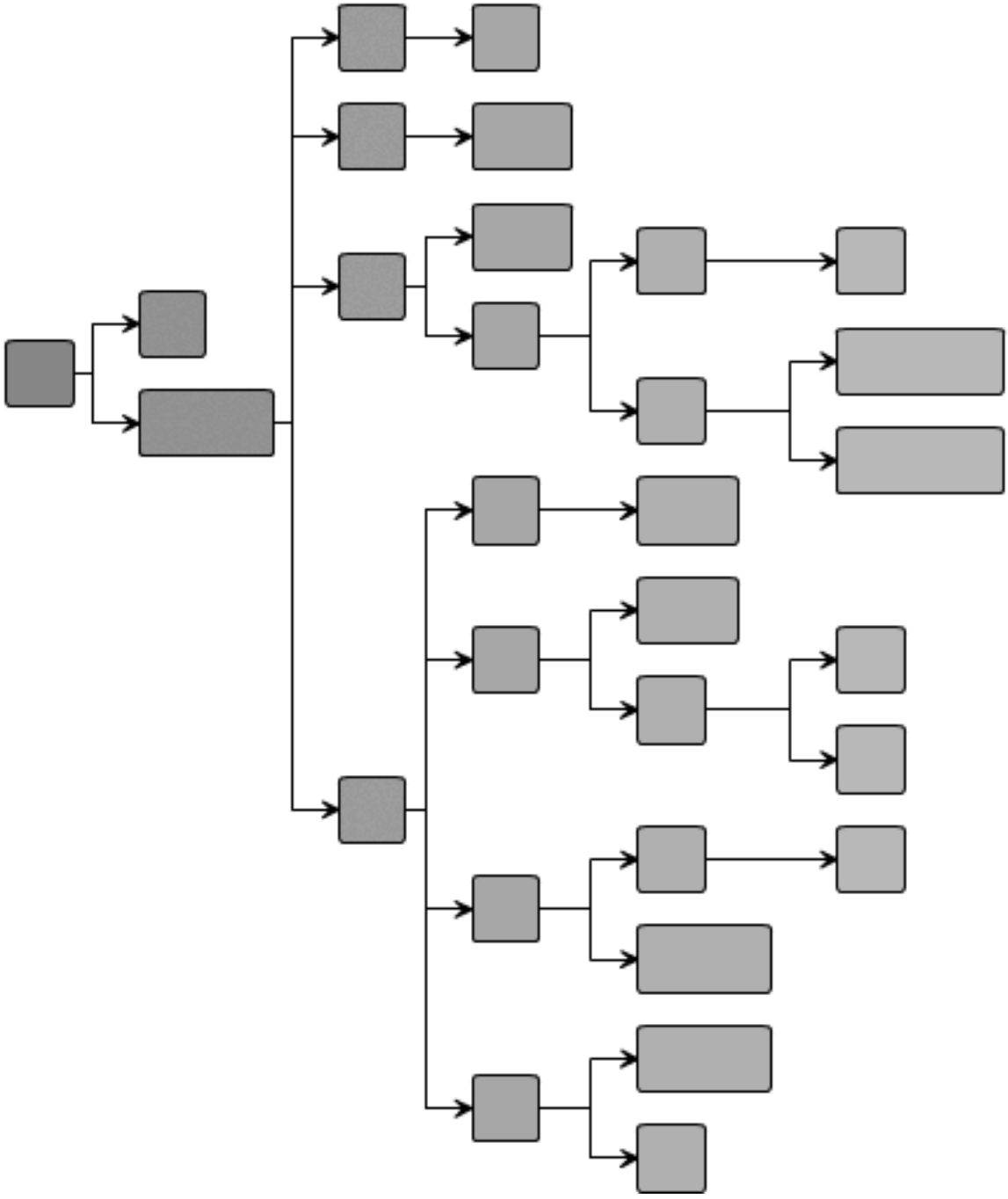
Graph 10: Hierarchic layouter (yWorks 2010)



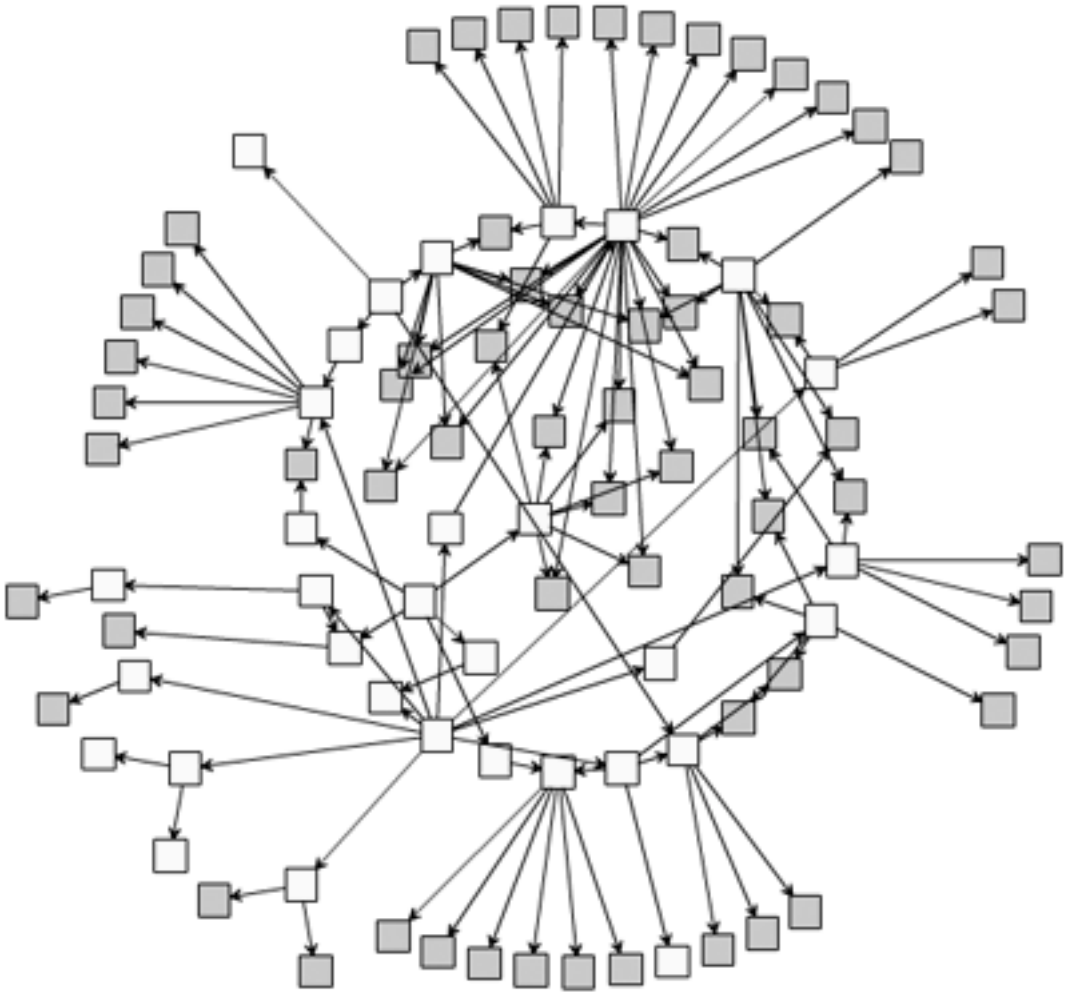
Graph 11: Organic layouter (yWorks 2010)



Graph 12: Orthogonal layouter (yWorks 2010)



Graph 13: Tree layouter (yWorks 2010)



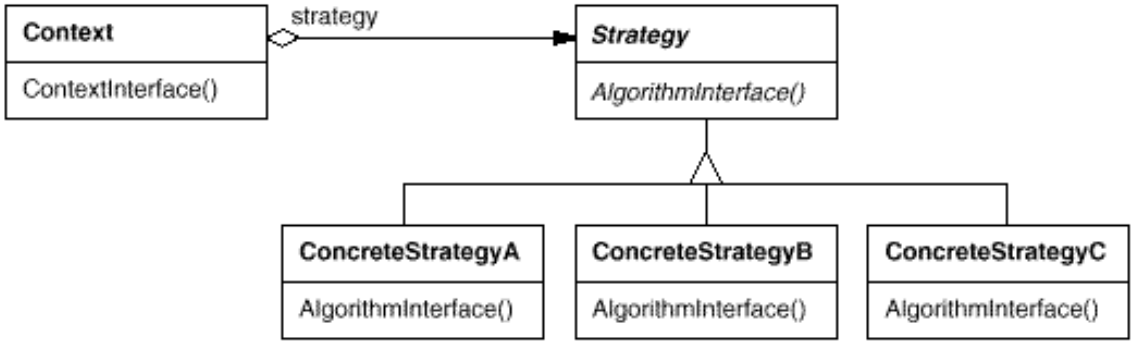
Graph 14: Circular layouter (yWorks 2010)

4.6 Design Patterns

The following subheadings provide a short description of the most important Design Patterns used in this body of work. Design Patterns are a way to create reusable object oriented software (Gamma, et al. 1994, 1).

4.6.1 Strategy

Strategy Pattern is one of the simplest Patterns out there. A Strategy Pattern allows the selection of an algorithm at runtime, i.e. Strategy Pattern allows one class to work in a number of ways by delegating the process of an algorithm to another implementation. (Gamma, et al. 1994, 315)



Graph 15: Structure of Strategy Pattern (Gamma, et al. 1994, 316)

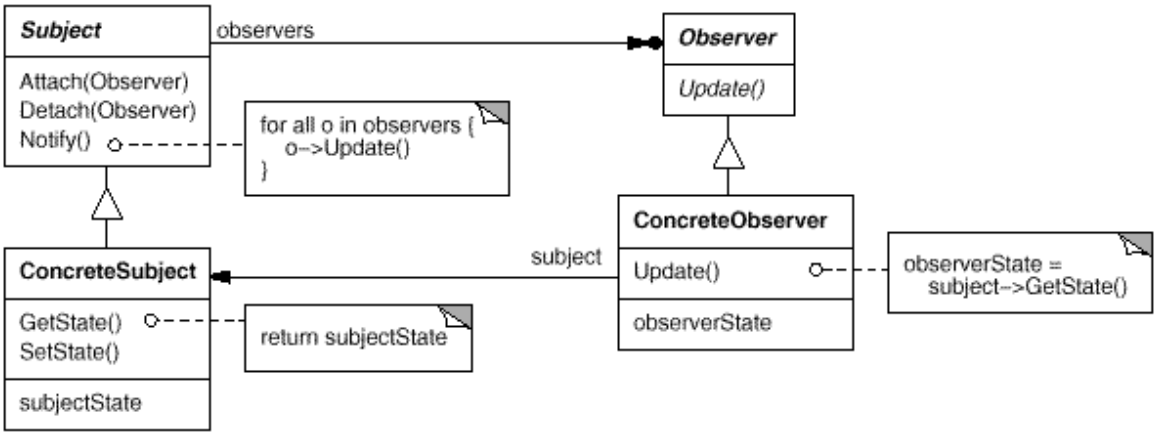
A strategy class implements an interface. Using this notion you can have a family of strategy classes which all implement that same interface. A class with a variable of that type of an interface can change the implementation of that variable to any one in that family of strategies.

For example, imagine a duck simulator application. This application models different kinds of ducks within it. Different kinds of ducks have different kinds of quacks. Some ducks have the same kinds of quacks though. To reasonably manage different kinds of quacks it would be sensible to define quacks as classes of their own. Then we could make all these classes implement the same interface (let's call it DuckQuack) and define in that interface a method called quack. Now we could associate a duck with a DuckQuack implementation and whenever a duck needs to quack we would delegate the call from the duck class to the DuckQuack implementation. If a duck becomes old and loses its ability to quack, we could switch to a DuckQuack instance which is unable to quack (i.e. the method in it does nothing). (Freeman, et al. 2004, 2-24)

The Strategy Pattern is used in many places – so many that it would prove to be rather difficult to list them all. It's a Pattern that you use even if you don't quite realize it. In practice every time you're using an object as an interface you're using the Strategy Pattern. Simply put, it's too arduous to go over all the places where this occurs.

4.6.2 Observer

The Observer Pattern is used to link together different objects. In this Pattern these objects are of two types, the ones that listen to other objects (Observer) and the ones who signal the other one to do something (Observable). I.e. when the state of one object changes all its dependents are notified of this change and they can then change their own state. (Gamma, et al. 1994, 293)



Graph 16: Structure of Observer Pattern (Gamma, et al. 1994, 294)

An Observable object keeps a list of observing members. This list is usually typed by some interface and it usually implements a method called update. Whenever the Observable changes its state it simply goes through all the registered members in its list and calls their update method. (Gamma, et al. 1994, 293)

A simple real world example: Imagine that there's a radio station broadcasting a message. There are some end-devices listening to that broadcast. Essentially, in

this rather mundane example, the radio station is the observable while the end-devices are observers. Whenever the radio station sends a message the end-devices pick it up and decide what to do with that signal.

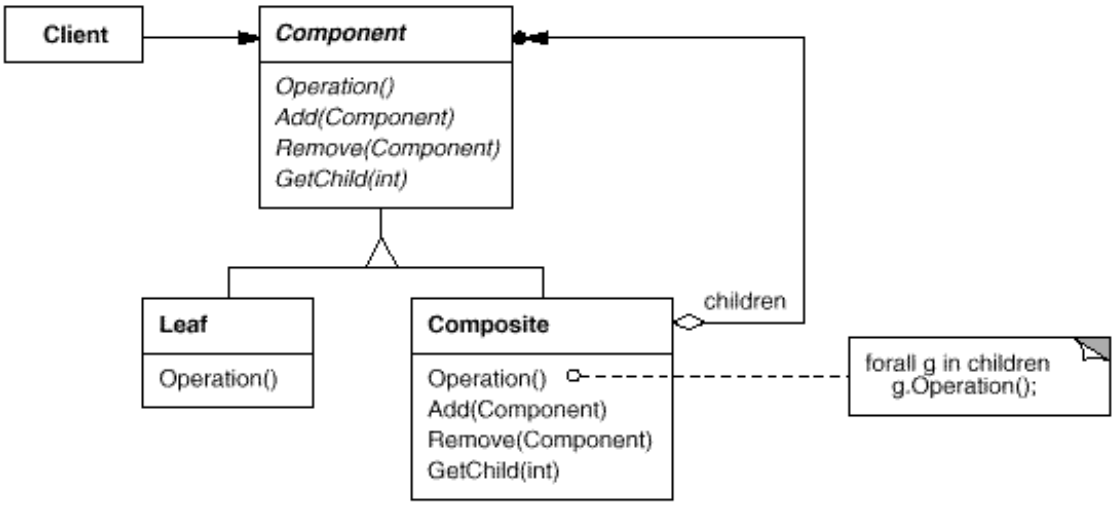
The Observer Pattern is an integral part of the MVC Pattern. Other than that, the Observer Pattern is not used very heavily in this work except when creating undoable commands.

4.6.3 Composite

The Composite Pattern is very much similar to the Decorator Pattern. The Composite Pattern essentially allows you to create a tree structure to represent hierarchies. The intended idea is that you could have the root object of that tree and only use it to manage all the subsequent nodes in the tree. (Gamma, et al. 1994, 163). This is not limited to this case however and you should use it as best suites your needs. You could also use objects randomly from that hierarchy because they all work in the same kind of manner, i.e. they all implement the same interface.

Instead of using just single objects in the composite hierarchy you can also use the entire hierarchy as a single object. It might sound that you would need to implement something to make this happen but you don't. Using the composite patter allows the person using it to deal with a clean and simple interface, while at the same time making it easy to extend your composite collection. (Gamma, et al. 1994, 166)

The setback is that you can easily create huge structures. Managing these huge structures can of course become cumbersome. But as with all good things, nothing is quite perfect. (Gamma, et al. 1994, 166)



Graph 17: Structure of Composite Pattern (Gamma, et al. 1994, 164)

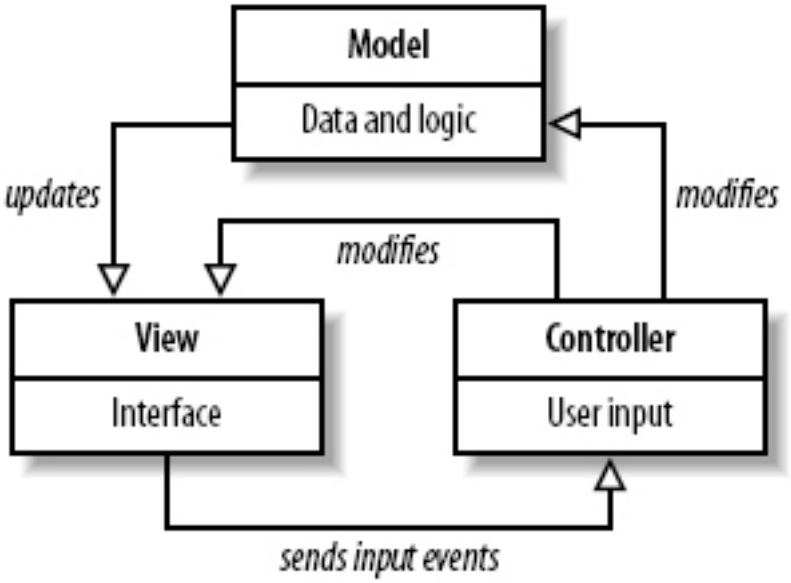
The proverbial example when discussing about the Composite Pattern is always UI components, especially Java Swing implements the Composite Pattern in an exemplary manner.

Example: On top of the Swing UI hierarchy is an instance of a JFrame. You can add various other components to this JFrame instance, like JButtons, JLabels, JPanels and so on. The JFrame is then responsible for managing the lower lever items found in the composite structure.

4.6.4 MVC

The MVC Pattern is not a single Pattern; it's a compound Pattern. This means that it is composed of a number of other Patterns, sewed together and made to look pretty in pink. The MVC Pattern is rather difficult to understand or to implement despite good real world examples. (Freeman, et al. 2004, 529)

The MVC Pattern composes of three major parts: the View, the Controller and the Model. The MVC Pattern uses three other Patterns to make it work: the Observer Pattern, the Strategy Pattern and the Composite Pattern. (Freeman, et al. 2004, 529)



Graph 18: Structure of MVC composite Pattern (Moock 2011)

Between the Model and the View an Observer Pattern is used. This is to enable the Model to update the view when it itself is updated. So, the Views register themselves with the Model to receive updates of its state. The Views then update themselves based on information they query from the Model when the need arises (when the Model signals an update). It's important to remember that the Model has no dependencies on Views or on Controllers - Views can simply register to listen to state changes of the Model. (Freeman, et al. 2004, 532-533)

Between the View and the Controller lies the Strategy Pattern. The View can choose the Controller it's associated with. This means that if the functionality needs to be changed in the future the View can simply swap its Controller to another one.

The View in itself is a GUI Composite. It means that you don't have to separately update each of the elements of the GUI component, instead you update the top level component and the top level component then deals with all the lower level components via delegation or whatever mechanism is used. (Freeman, et al. 2004, 533)

Perhaps the most evident example of the MVC Pattern is the Web browser. The Web browser renders the view on the screen based on HTML it received from a Web server. This HTML code is basically the Model of the Web browser. The View part of the Web browser is what you see on the screen after the browser has finished parsing the HTML file and has formed a visual representation of it.

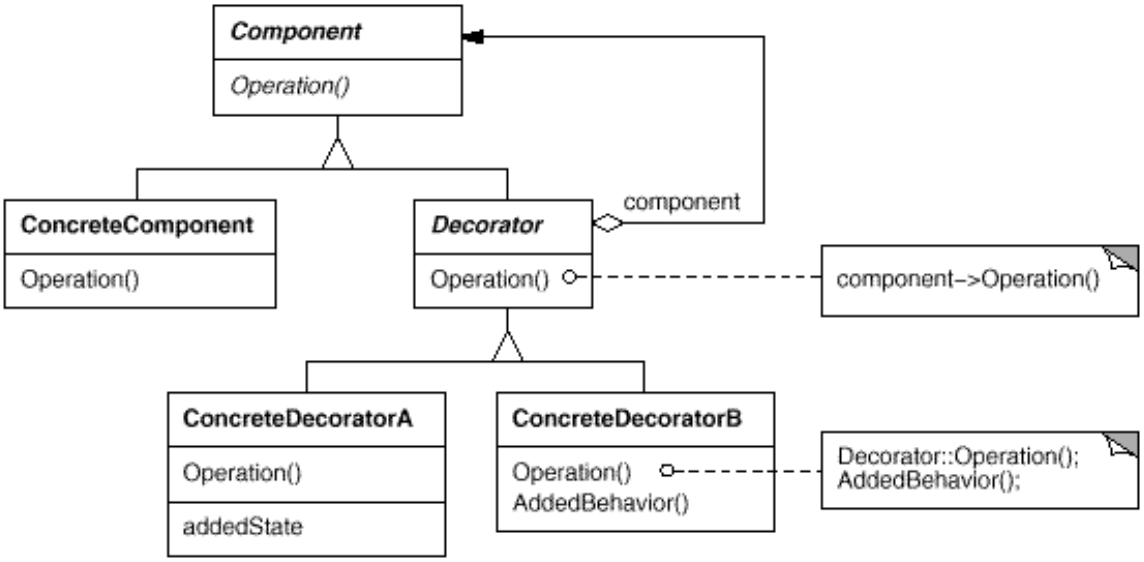
In this context the Controller part is the address bar where you can input an address for a web page, thus changing the underlying model. JavaScript is also a controller for a page (could be some other script language as well), which is used to alter the state of the HTML code. For example with JavaScript you could remove elements or add new elements to the HTML code. The browser then updates the View automatically.

MVC Pattern can be found from almost anywhere where user interfaces are built. In this body of work I didn't use it myself but the yFiles Java library uses it and it's good to understand how a tool works.

4.6.5 Decorator

To separate responsibilities among different classes, rather than a single class, the Decorator Pattern can be used. The Decorator essentially forms a chain of objects, each object decorating previous one in the chain. The Decorator can also be understood as a wrapper class. (Gamma, et al. 1994, 175-184)

All the Decorator classes share the same interface or base class. The Decorator class has a member object of that interface, essentially wrapping it in itself. (Gamma, et al. 1994, 175-184)



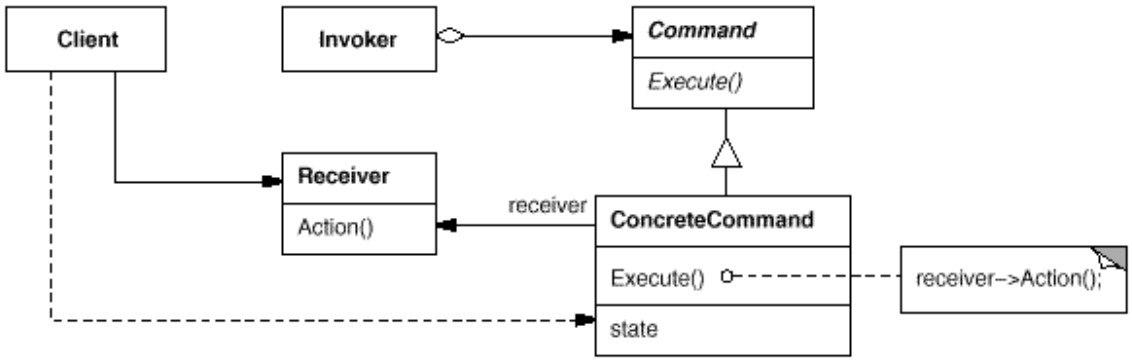
Graph 19: Structure of Decorator Pattern (Gamma, et al. 1994, 177)

For example, a template parser works on text file. It parses the text file and whenever it comes across a keyword which signal an action to take, it delegates that keyword to an object that knows what to do with it. This object can be implemented as a Decorator where each object in the chain is responsible for certain kind of behavior. For instance, if the parser comes across a piece of text "[insert_current_time]" it recognized it as a keyword because it is enclosed within brackets. It then inputs that keyword to a decorated keyword-handler object, which is composed of a number of decorating objects. One of these objects knows how to deal with "[insert_current_time]" and successfully returns the current time for the parser.

The initial idea to use a Decorator for handling different kinds of graph plotting strategies came into mind. It's something that might not be in the final design at all, but there's a good chance that it will be mentioned.

4.6.6 Command

To issue commands in an application is a rather frequent operation. Usually you hard code these commands into the application, essentially predefining what happens every time you execute a certain routine. The Command Pattern is created to allow you to change that command on the fly, without recompiling. The Command Pattern encapsulates a command to an object. (Freeman, et al. 2004, 206-207)



Graph 20: Structure of Command Design Pattern (Gamma, et al. 1994, 236)

To employ the Command Pattern all you really need to do is to create an interface, which is implemented by all command objects. You then program your application against this interface rather than implemented objects. This interface defines methods necessary for your application – usually it's enough to define a method called execute, which is essentially responsible for creating the functionality defined in a given command object. (Freeman, et al. 2004, 207)

Sometimes you also want to be able to undo commands you've executed. This is a nice feature for any application. To extend your command objects to support undo you only need to extend your command interface to house another method called undo. In case of undo methods you do however need something to keep track of undoable commands. This can be anything you want it to. Easiest thing to do is to create a singleton object running in a separate thread, responsible for keeping

track of executed commands and route all your undo requests through this singleton object. (Freeman, et al. 2004, 216-220, 228)

4.6.7 Singleton

Singleton is probably the most used Design Pattern in software industry. It's also probably the simplest one. The responsibility of the Singleton Pattern is to ensure that there is only one instance of a given object used in the application. (Gamma, et al. 1994, 127-134)

Singleton composes of only one class and its functionality is pretty straightforward. The Constructor is declared private, preventing any other object or class except the class itself to instance new objects. The class has a field holding it's own type; this is essentially the only one object within the whole application. The class also declares one static getter method, which is responsible for returning the singleton instance to the caller. Usually, if the singleton instance is not defined, it is defined in this method only once. (Gamma, et al. 1994, 127-134)

Consider a logger object. This logger object is responsible for managing all logging operations in the application. To effectively store data and possibly even display information, it is prudent that there is only one logger object present in the application. If you have more, then you'll have to deal with concurrency issues when they're writing information to file or creating log files. Singleton fits this bill perfectly.

Singleton is again something, which is used extensively in this work. The places are rather numerous. There's a logger and there's going to be a number of instances, which handle threads and so on.

4.6.8 Memento

The Memento Pattern is used when a state of an object needs to be saved. It would be cumbersome to save the state to object itself, rather it's easier to use another object simply for keeping track of all it's defining attributes. Whenever an object needs to be restored back to its previous state you can simply retrieve this state from the memento object. (Freeman, et al. 2004, 624-625)

To create a memento class you need to simply account in it all the important attributes found from the object, which state it is saving. Using a memento object can prove to be difficult if your application is in a constant state of change because every time you change something in the original object you also need to accommodate this change into your memento object. (Freeman, et al. 2004, 624-625)

4.6.9 Façade

The Façade Pattern defines a rather simple idea, i.e. to provide a simple programming interface to a complex system. The façade allows you to use complex composites through a single object making your system easier to use. (Freeman, et al. 2004, 264)

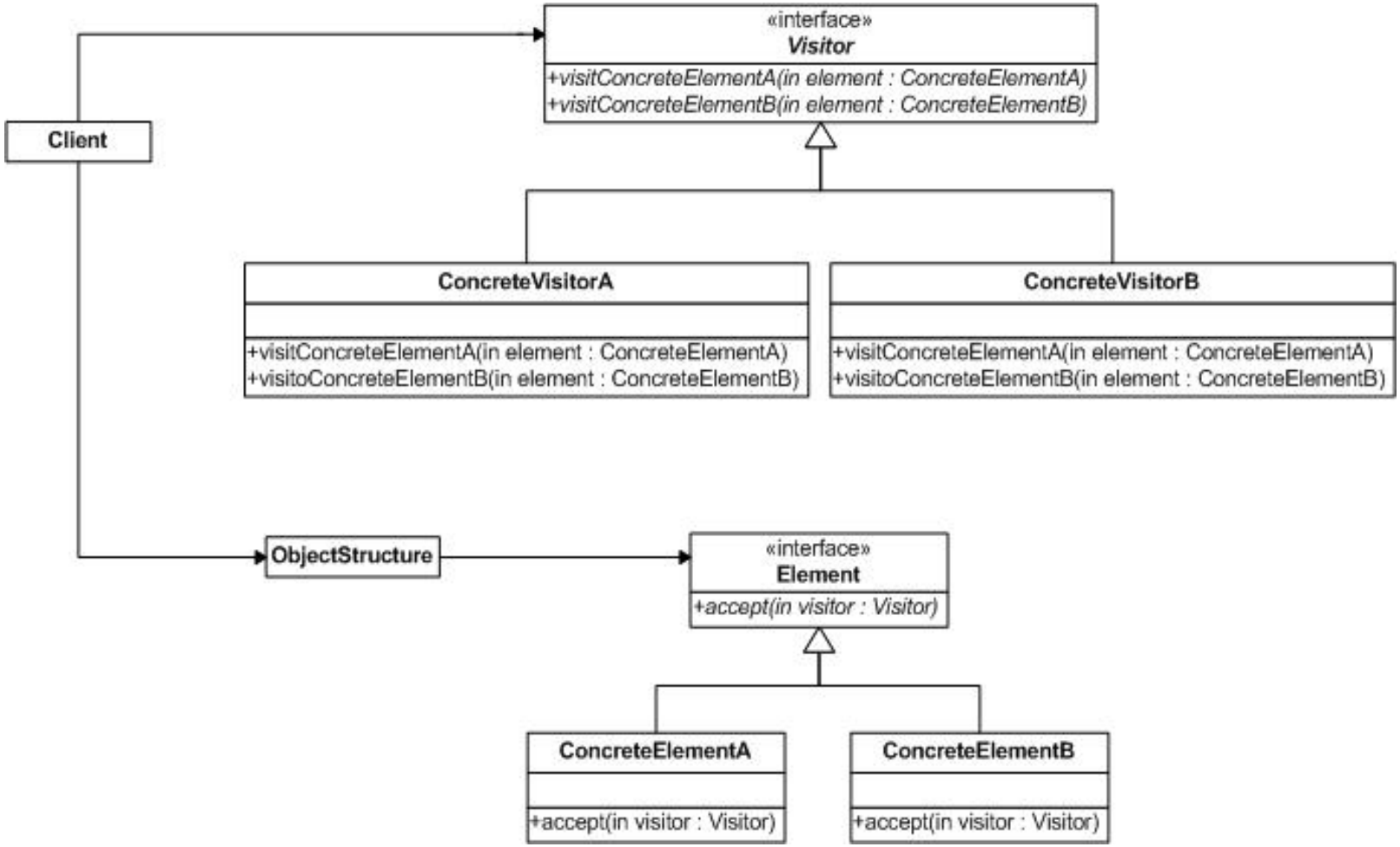
4.6.10 Visitor

It's as ominous as the lizards in the TV-show. Yet it's kind of nice to have around in a terrarium. It's scary and it's a commodity item all wrapped up into one. Who wouldn't love that?

The Visitor Pattern is used with composites to extend their functionality. The Visitor Pattern allows you to traverse a composite collection. An important aspect to take note of when using the Visitor Pattern is that it breaks encapsulation. (Freeman, et al. 2004, 628-629)

The Visitor allows you to go through a group of objects that belong to the same family. A visitor object visits each object in turn, inspects their state and does something with that information and possibly even alters the state. The visited object essentially exposes its internals to the visitor object and this is the only drawback of the Visitor Pattern. (Freeman, et al. 2004, 628-629)

On the next page is a class diagram of the Visitor Pattern.



Graph 21: Visitor Pattern structure. (Gamma, et al. 1994)

Implementing the Visitor Pattern is a rather messy business mainly because it involves spreading code related to one group of classes in various places. It's easy to get lost after a while. The Visitor is an appealing choice but it also brings trouble to the table. (Freeman, et al. 2004, 628-629)

4.7 Software Engineering

The primary concern when choosing a software engineering principle is to adhere to good object design. The idea is to create a maintainable solution, which will serve to support future development of the system. (Larman 2005, 3)

The goal is to provide clear Object-Oriented Analysis and Design (OOA/D), using Design Patterns and to effectively communicate these intentions in UML (Unified Modelling Language) format. What lead to good OOA/D is good Requirements Analysis, principles and guidelines of software development. All this work is done in a iterative fashion with agile Unified Process (UP). (Larman 2005, 5)

I'm first going to describe all these things in brief before I describe exactly how they were employed.

4.7.1 Agile over Waterfall?

Agile software development is something, which emerged after the waterfall process was found to be ineffective. The waterfall process places a lot of weight on documentation and especially documenting everything before anything is actually coded. This ideology proved to be disastrous in many projects, although it can actually work with a lot of problems. Choosing whether to use Agile over Waterfall is largely just a decision made before the project is launched and there is no real guideline to choose one over the other. However, these days the Waterfall

process is largely frowned upon and in general Agile is chosen over the Waterfall process. (Larman 2005, 23, Vandenburg 2010)

To support the claims that the Waterfall process is inferior to Agile process, research display the following numbers:

> On average, 45% of the features in waterfall requirements are never used, and early waterfall schedules and estimates vary up to 400% from the final actuals.
>
> In hindsight, we now know that waterfall advice was based on speculation and hearsay, rather than evidence-based practices. In contrast, iterative and evolutionary practices are backed by evidence - studies show they are less failure prone, and associated with better productivity and defect rates. (Larman 2003, Larman and Basili 2003)

Furthermore, the waterfall process was faulty from the beginning. It was not supposed to be implemented as it was, in fact it was implemented in a completely false manner. The waterfall process was build upon a paper by Winston W. Royce. The paper was misinterpreted however because it didn't try to explain how software should be engineered; it tried to explain how software should not be engineered. For some reason though, it ended up being used as a guideline for software engineering. (Vandenburg 2010)

When first creating guidelines to software engineering, a task force discovered viable solutions, which describe today's Agile approach rather identically, i.e. they defined that software should be engineered in an iterative fashion with high emphasis on testing. A few years later the waterfall process was created. It was an attempt to mimic software engineering with traditional engineering, like bridge building for instance. Traditional engineering does not however play nice with software engineering, which is completely different in almost every aspect. (Vandenburg 2010, Larman 2003, Larman and Basili 2003)

To defend the Waterfall process one has to consider that the time when Waterfall was used was a time when access to computers was limited. At this time a developer could possibly only get a few hours of computer time per week.

Considering this point of view, good design was more important than testing. This goes on to imply that with contemporary tools the Waterfall process is not a reasonable choice (Grzegorz 2011).

Based on these researches and results found, I chose to use an Agile process rather than a Waterfall process.

4.7.2 Agile Unified Process (UP)

Agile is an iterative and evolutionary process where a problem is not presumed to be ready until it is actually coded and found to be good. Rather an initial design is produced, in the fashion of Agile Modelling (more on Agile Modelling in the next paragraph), and implemented as soon as possible. Feedback from the coded version of the model is then used to enhance the design to support any possible disabilities encountered. (Larman 2005, 17-22)

In Agile UP, Agile Modelling is used. This means that quick sketches of a design are produced in the easiest possible way (Larman 2005, 30-31). For me this meant to draw them out on a piece of paper. In this work I'm only including computer reworked images and only when it suites to illustrate a point. I sketched a lot of class and sequence diagrams. To include them in this report would only serve to confuse things (plus I already threw away large number of the sketches).

4.7.3 UP Project Phases

UP is a software development process used for building, deploying and maintaining software. It is an iterative process, like all agile processes are. (Larman 2005, 33)

Four major phases are included in a UP project. These major steps hold within them a number of iterations in which whatever might come along is tackled in detail, but this is the big picture of a UP project. (Larman 2005, 33)

1. Inception: approximate vision, business case, scope, vague estimates
2. Elaboration: refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
3. Construction: iterative implementation of the remaining lower risks and easier elements, and preparation for deployment.
4. Transition: beta tests, deployment.

All the different parts of a UP project – Inception, Elaboration, Construction and Transition – are iterations. These iterations base weight on different aspects of a project. The Inception stage of a project is meant to roughly define what is to be done and also to include proof-of-concept. Elaboration is a step to further define the Inception stage. In Elaboration it would be desirable to further elaborate on requirements and to create a core model for the application. In Construction stage the application is built on top of the core model, which was designed and implemented in the Elaboration stage. Transition deals with beta testing and deployment. (Larman 2005, 33)

4.7.4 UP Disciplines

This chapter is only included for clarifying the UP Phases. Not much attention is paid later on to these issues, rather they are allowed to evolve and progress as best deemed in any given situation.

UP disciplines which are important in this body of work are:

- Business Modelling
- Requirements
- Design
- Implementation

All of these are included in a UP Phase iteration, and the disciplines merely describe the problem area where we're currently working on. For instance in a UP

Phase iteration you might start with creating a business model to support the requirements you are going to create. Next you would implement those requirements to a design and maybe even execute them if you would deem them important enough to take precedence of things, which have been created earlier. (Larman 2005, 35)

4.7.5 UP Artifacts

In UP, an artifact is any kind of a document in the project. This includes within itself all diagrams produced, text documents and basically everything done for and in the confines of this project. (Larman 2005, 34)

The important thing to note about the artifacts is that one has to choose the artifacts he is going to include in his work. This does not of course mean that you should make this decision in the beginning of the project and then live with. No, you can include additional artifacts to the project as it progresses, which is very Agile-like. (Larman 2005, 34)

The primary artifacts used in this work are Class diagrams, Sequence diagrams, use cases and some icons and images. Here and there you can also find artifacts like “Vision and Business Case”. These are artifacts, which describe an important, software project centric, requirement. Whenever this occurs, I’ve dropped in the text a short text describing that this is an artifact. Images consist largely of UML sketches.

One could also create an artifact, which would describe all the artifact types used in the project. This however seems largely unnecessary for this work. (Larman 2005, 34)

4.7.6 Use Cases

Capturing requirements for a software project is difficult. There are however good tools to work with. One of the most useful tools lying around are use cases. They are effective because they're easy to understand (Pressman 1997, 608). This essentially means that it's easy for even the most non-technical person to create a use case. (Larman 2005, 64-65)

A use case simply dictates how a user might use the application. A typical use case consists of an actor and a scenario (also known as a use case instance). The actor is essentially the end-user, while the user can play various roles, i.e. act as different actors, of the application and the scenario describes how the actor interacts with the application. (Larman 2005, 61-63, Pressman 1997, 608)

There's really no official form how a use case should be written. As long as it defines functionalities that should be present in the application, it is considered sufficient (Pressman 1997, 608). There's three ways to categorize use cases: brief, casual and fully dressed. Use cases found from this body of work are mostly casual. A casual use case is something, which simply quickly dictates the issue in hand; it doesn't impose a structure. I worked the more important ones to a stripped down version of a fully dressed use case. This, too, is just a template and I've varied it occasionally to suite a scenario better. Table 1 below illustrates this stripped down structure. (Larman 2005, 65-79)

Table 1: Use case template

Use Case Section	Comment
Use Case Name	Short, informative name.
Preconditions	What needs to be set before this scenario can occur?
Main Success Scenario	Describes the main flow of the scenario. I.e. a succession of steps that occur when the user initiates an action.
Exceptions	What might cause exceptions in the scenario?
Frequency of Occurrence	How often this scenario is likely to occur. Possible

	values are: seldom, frequently and continuous.
Miscellaneous	Any additional remarks concerning this scenario.

5 TOOLS OF THE TRADE

A number of technologies are utilized in this body of work. This chapter provides a brief summary of all of the essential technologies. I'm not going to go into details about operating systems. Mainly I used Windows Vista and Windows 7 but because most of the work was done in Java, the operating system used doesn't hold much weight because it does not really matter what operating system you are using.

5.1 Java

Java is both a language and a platform. The Java programming language is an object-oriented language, which means that everything in Java is an object. The Java platform is essentially a virtual machine, which works on top of an operating system. This platform, the Java Virtual Machine (JVM), is responsible for running compiled Java byte code (compiled from the Java language source files of course). (Oracle 2010)

It is worth to mention a few details about Java. Java, the core of it, as such doesn't offer many libraries when it comes to UI's. However, Java offers extensions (usually included within a Java distributable), which introduce UI libraries. The primordial Java UI library is called AWT. AWT has been replaced, or extended with, Swing. Swing is the essential tool for building UI's with Java.

5.2 Qt

Qt is advertised as a cross platform class library, primarily for C++ applications. It offers an extensive UI library to be used with a number of languages. Qt differs from Java in many ways. First of all, Java is a platform whereas Qt is simply a UI library. Qt would correspond better with Java Swing rather than full-blown Java.

Qt works with native applications, i.e. you write and compile the code you create for a specific platform. (Nokia 2010)

5.3 External Java Libraries

Two external Java libraries are used in this work: yFiles for Java and fastutil. yFiles for Java is the single library which, makes the creation of this thesis work possible. It's the graphing library used, which offers algorithms for graph analysis and visualization. (yWorks 2010)

fastutil library extends the functionality of Java Collections Framework. It essentially creates prebuilt versions for all primitive data types concerning collections of items. Without this library you would have to use composite data types which of course consume more memory in a given application. (fastutil 2010)

5.4 Java Native Interface (JNI)

In this work it was necessary to interface a Qt application, i.e. a C++ application, with Java. There exist a number of tools for this such as Jambi and Jace but after researching a bit and doing some tests with these systems I opted to use the original and well-proven technology of JNI. The other tools offered nice features, which would make a developers life easier, but for my purpose, which was quite simple in the end, all this didn't mean much.

JNI essentially enables the integration of C++ code with Java code. E.g. to build an interface from C++ to Java you would create a JVM inside your C++ application and in that JVM you would start the Java application you choose. JNI then offers you with "hooks" that you can use from C++ to tell the JVM what to do. (Oracle 2010). It's really a quite nice system if you're interested about bridging technologies.

5.5 Concurrent Versioning System (CVS) for Version Control

Good software engineering policies dictate that you should always use a version control system (VCS) in your projects. I chose to use CVS because of various reasons. New, more robust, tools exist but CVS is quite adequate for version control purposes.

CVS is an old system; it's one of the first version control systems in existence. Using CVS (using any version control system in fact) you can record the history of files and documents. (CVS 2010)

5.6 Integrated Development Environments (IDE's)

A number of IDE's were used in this work. The most important of these were IntelliJ IDEA for Java development and Qt Creator for Qt application creation. I used Microsoft Visual Studio 2010 for testing C++ graphing libraries. Eclipse was used just for the same purpose. I also thought about using Eclipse IDE instead of the Qt Creator but didn't stay with this decision in the end.

The number of IDE's used might seem a bit too much but it really was necessary to look under every rock when creating interfaces between C++, Java and the ALMA server. Of course you could've done all the things in one IDE but it offered some perspective to use different tools considering that my personal experience with creating C++ applications was next to nothing – still is.

6 LEG WORK

6.1 Inception

Artifact: Risk List & Risk Management Plan:

The initial objectives for the project are to implement the visual graph-like structure, described in chapter 2. Concerning Java there really is no big problems immediately in the horizon. Concerning Qt the problems are vast.

Since the primary aim of this project is to provide a working application for Java, I didn't want to slow down the start by concentrating on Qt. So, at this point I already decided to tackle and create the project first in Java (at least I would finish the elaboration phase before I would move on to Qt); I would create the Ot application after I had successfully modelled and coded the core model for the Java application. It seemed like a prudent choice to concentrate on one thing at a time and leave everything relating to Ot to a later stage.

This decision was made because I did a little bit of research on Ot and found the lack of a proper graphing library (which would be able to produce tabularized views of a graph for schematic purposes) as something, which would require a lot of research work to get even started on. And to be honest I was anxious to get to do something other than research.

The next big problem on the list was the implementation of tabularized schematic views in Java. This, however, was something, which could be quickly researched from yFiles documentation. I found it to be doable even though it would take great many hoops to jump through.

6.1.1 Initial Requirements For The Project

Artifact: Vision and Business Case:

The application is able to visually represent the structure of the ALMA system with all it's objects and links. Additionally it is able to represent schematics. These representations should be incremental so that the user is not presented with all the objects in the database all at once. Rather the user chooses a point of origin and then begins to investigate it further by interacting with the application.

The application needs to differentiate the objects described so that it is visually clear of which kind of an object the user is inspecting. This basically means that an icon should be portrayed with the object in question. This icon is something that is used as a de facto standard in the ALMA system.

The connections between objects can be, and most often will be, directed. This means that a link displayed between two objects needs to be able to represent, which object is the linker and which one is the linked object.

The user is able to define what he wants to display in the graph. This means that the user is able to define if he wants to see the links for instance, or if he wants to see links only of a certain type, etc.

The user is able to specify how objects are grouped and what objects should be abstracted and also which objects should offer interactive abstractable behaviour. Interactive abstractable behaviour is when, for example, you have a group of wires, which have been abstracted as a cable, and you would like to be able to expose the abstracted objects or vice-versa.

Artifact: Prototype, proof-of-concept for Java: Demo Application. This is considered sensitive material and has thus been censored from this work.

For Qt there is no proof-of-concept, since merely the proof-of-concept for Qt would take a lot of effort. Instead the first Elaboration steps concerning Qt are used to provide proof-of-concept for the key areas of the application. This is a wrong way to use UP but since pretty much everything is allowed within the confines of a UP Agile project I considered this to be my prerogative.

6.1.2 Use Cases - functional requirements

Artifact: Use-Case Model

You can find a list of use-cases used with inception step from appendices. As defined by the UP process, these use cases are largely not yet defined other than by name. Only a small subset of them is exhaustively documented. These use cases have been extracted from Initial Requirements (chapter 6.1.1) as well as from interviews. These use cases also include a few personal "nice-to-have's" as well.

6.1.3 Implementation

Implementation at this point didn't cover much work. Mostly it was testing the yFiles library and seeing how it works - nothing too exciting. At this point serious implementation would be a waste of time because the requirements are hardly executable at this point.

6.2 Java Elaboration 1

This step builds on top of the Inception phase, described in the chapter before this one. The main goal of this step is to further redefine and elaborate on risks and requirements. The first draft of the application core is also modelled and coded.

Table 2: Java Elaboration phase backlog

Priority	Requirement
1.	Core architecture
2.	Extend yFiles to accommodate ALMA system
3.	Core model

6.2.1 Tasks

All three items in the backlog are tackled in this Elaboration step.

6.2.2 Core Architecture

The core architecture of the program is responsible for identifying the different aspects of creating a graph from the ALMA system. First and foremost, the yFiles library needs to be extended to accommodate ALMA-centric features.

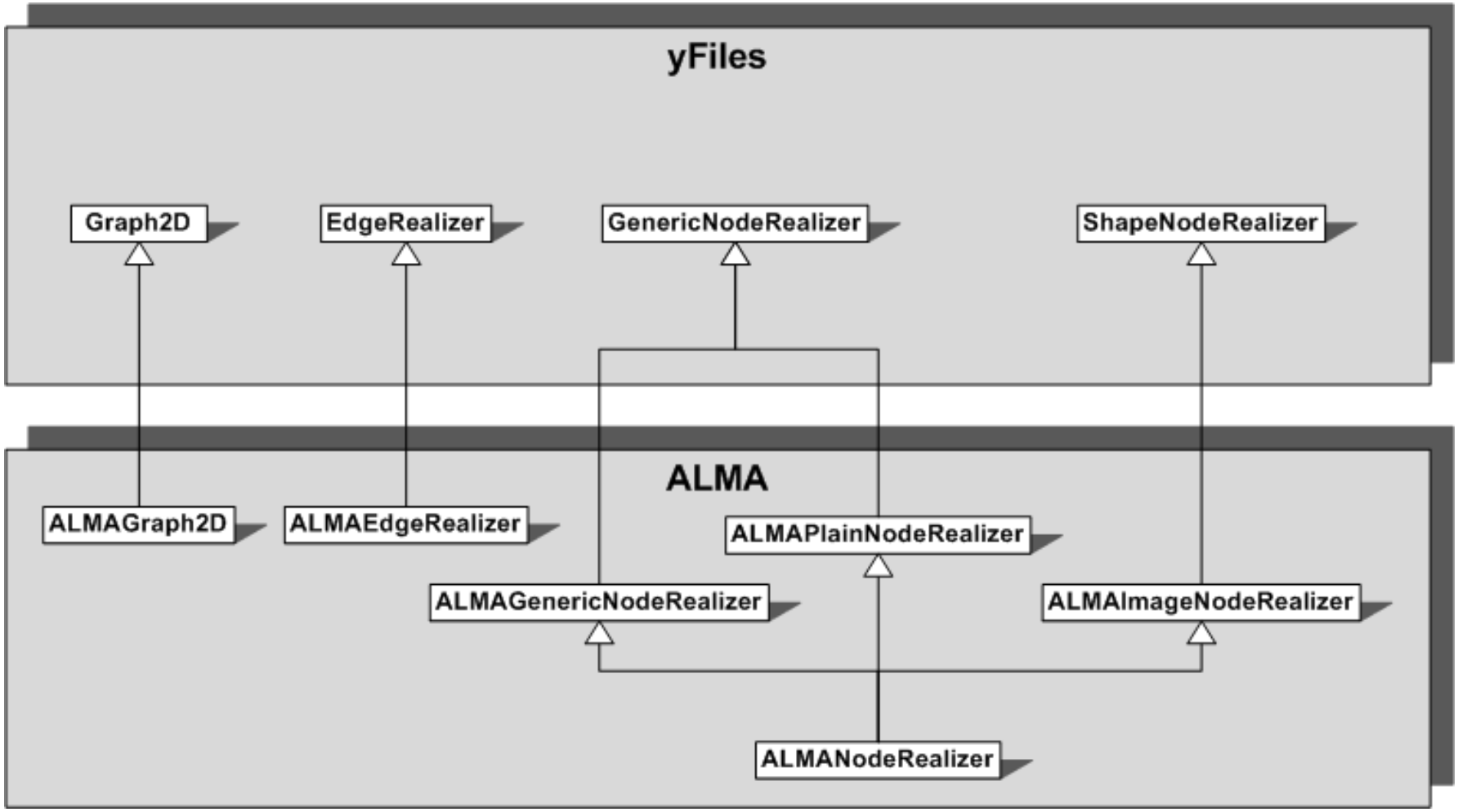
The different aspects of graphing fall into the following categories:

1. Extracting information from the ALMA system.
2. Input extracted information into the yFiles library, i.e. plot the graph.
3. Define the layout algorithm to be used.
4. Define the scope within which the graphing algorithm works in.

6.2.2.1 Superimposing ALMA layer on top of yFiles library

To properly incorporate the yFiles library into the ALMA system, it needs to be extended in some way and a proper separation of concerns needs to be figured out.

To extend the yFiles library is a rather straightforward procedure. All you need to do is extend the key classes in the yFiles library and superimpose your functionality on top of it. The following class diagrams describe the most important aspects of this inheritance structure.



Graph 22: ALMA layer on top of yFiles graphing library

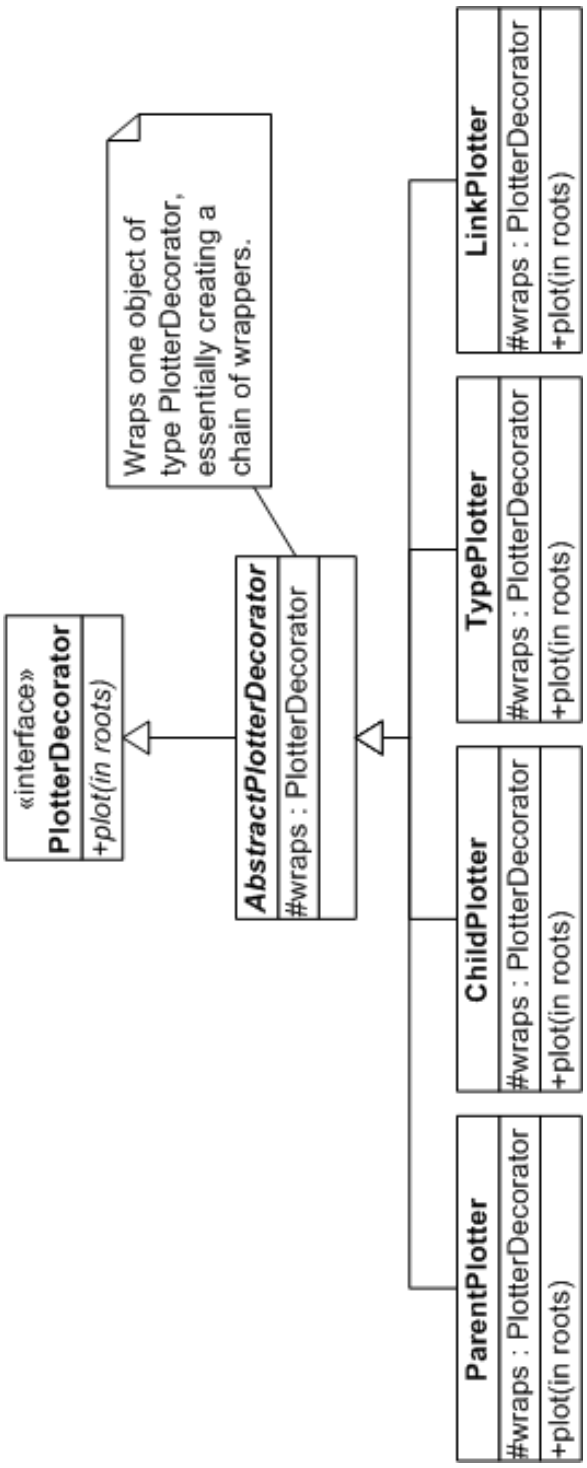
6.2.2.2 Core model

Definitely the hardest part of creating the core application is to implement different kinds of plotting solutions and how exactly they should be implemented.

Plotting is the process of extracting information from the ALMA system database and figuring out how to use that information. This is essentially the step, which separates whether the graph in question is a hierarchical view, a network view or a schematic view.

Based on sound judgement, i came to the conclusion that to begin the design process of the graphing part, it would be best to choose to use the Decorator Pattern. Essentially this would allow me to comprise the plotting algorithm from various algorithms, implementing (or including) a plotting scenario if it would seem prudent.

Following class diagram describes the core model of the system.



Graph 23: Core model, first version

6.2.3 Implementation

The implementation of this model proved to be a difficult one. The Decorator Pattern for plotting the graph doesn't work that well in the end for this purpose. A better solution could be to just create a stand-alone algorithm for different kinds of plot types (i.e. network, schematic, etc.). The problem with this approach is however that it would make the tool less generic and thus new application areas might need to do more work to make use of the tool.

6.3 Java Elaboration 2

The main goal for Elaboration step 2 would be to change the decorator of the plotter to something else.

6.3.1 Tasks

The only task planned for this iteration is to redesign and reimplement the applications core architecture and how it should exactly function. In iteration 1 it became evident that the Decorator Pattern didn't suite the need.

6.3.2 Core Architecture

Since the Decorator Pattern proved to be unsuitable for this purpose, a different Design Pattern is employed. If this Pattern doesn’t offer a workable solution, then a standalone solution will be employed. Reflecting with the Decorator Pattern, a somewhat similar approach is to use the Composite Pattern. The Composite Pattern is in many ways similar to Decorator Pattern. It does however offer some more flexibility concerning structure.

See section 4.6.3 for a detailed description of the Composite Pattern.

6.3.3 Implementation

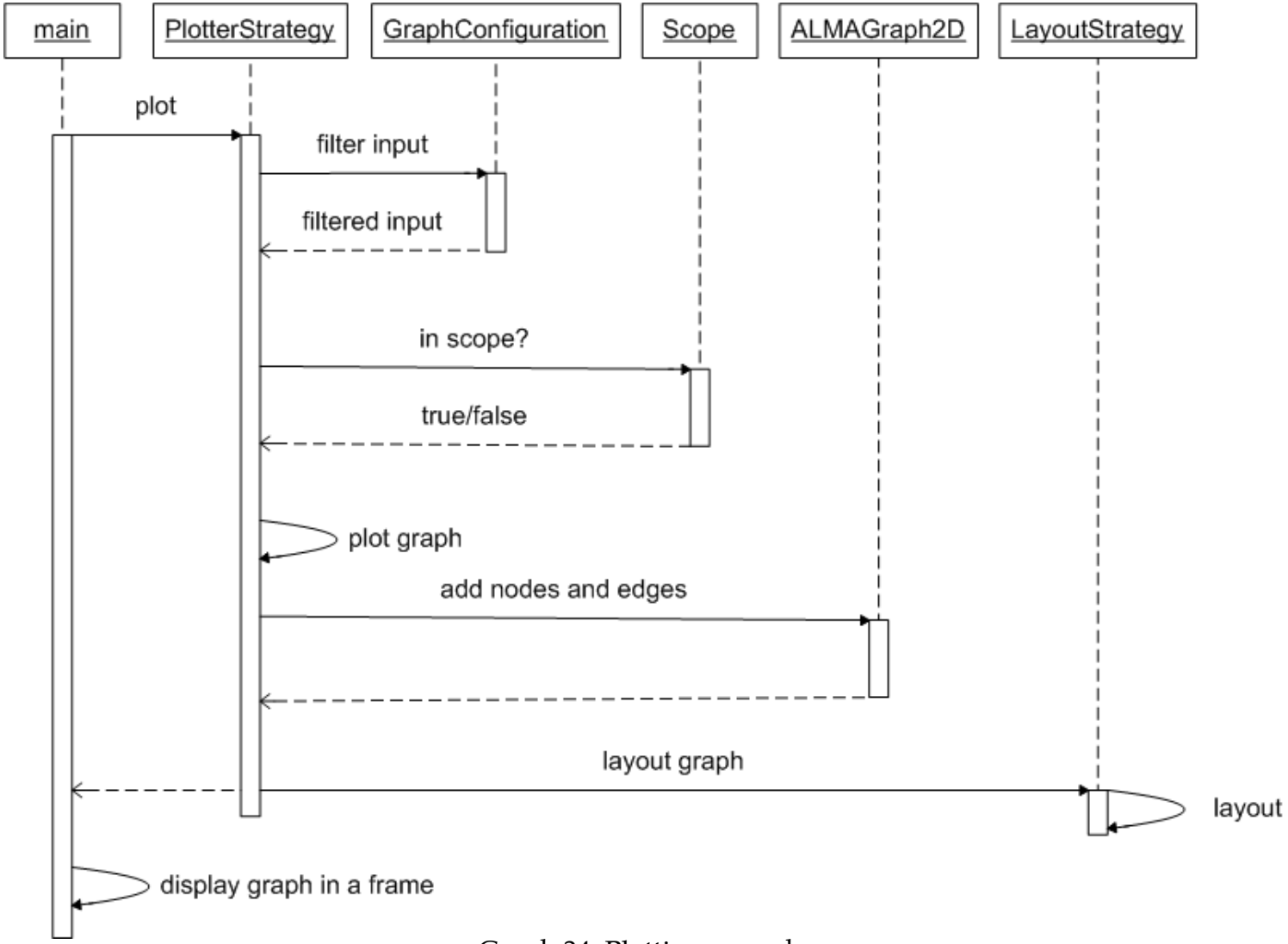
I didn't even get around to implement the Composite Pattern wholly. Somewhere between designing and coding it became evident that the Composite wouldn't offer much more than the Decorator Pattern did. Instead of trying to make it work, I decided to drop the design altogether and maybe figure out an alternative approach to the problem.

6.4 Java Elaboration 3

Since it's difficult to work this problem to a Design Pattern, a better solution would be to redefine the problem.

6.4.1 Core architecture

The problem is to effectively implement a plotter, which would be easily customizable. Trying to work this problem to a Design Patter didn't do any good. A different approach is to work directly with the ALMA system building blocks. The plotter essentially works with objects, figuring out their characteristics. To effectively customize what a plotter considers to take in would be to filter an object before basing any plotting on it. This way different kinds of plotters can plot everything without altering their structure to accommodate different kinds of scenarios. The plotter doesn't know, nor does it care, what are the "real" contents of an object are, it just works with the filtered version of the same object. The sequence diagram below illustrates the different things that a plot operation entails.



Graph 24: Plotting a graph.

6.4.2 Implementation

I created a filter utility method, which is responsible for filtering objects against user-defined settings. This works in a pretty straightforward manner. Essentially the user defines what he wants to see per type basis. This offers plenty of room to customize different views.

Despite the straightforward manner in which the solution works, the actual coded implementation proved to be arduous to create. It takes many LOCs (Lines of Code) to take into account all the possible definable configurations. However, this solution is easy to work with. Some issues might rise somewhere in the future when this tool would be used to input data as well, since a filtered object does not have all the same things in it that the original does. Thus it might create some ambiguities. But for now the approach seems to fit the bill perfectly.

6.5 Qt Elaboration

The Java application elaboration stage is now all done. What remains to be done is to pick-up on speed concerning the Qt side of building the same application. This part of building the application is still largely a mystery. What should have been done, before getting this far in the thesis work, would have been to do more research on Qt and its abilities to create graphs. However, since researching all these things takes a lot of time—time that I would have been forced to take out of the main goal of this thesis work, namely creating a working solution for Java—I decided to postpone everything relating to Qt.

This iteration is responsible for further researching Qt and its abilities. There are going to be minor implementations as well – small tests to prove the concept of interfacing Qt with Java and to test how different graphing applications work.

Table 3: Qt Elaboration phase backlog

Priority	Requirement
1.	Interface Qt (C++) with Java
2.	Once an interface to Java has been built, another interface needs to be created to connect to the ALMA system
3.	Research graphing libraries for Qt

6.5.1 Requirements

The overall end-requirements for the Qt application are the same as with the Java application. The Qt application is much more challenging in a number of ways though. First of all, to build a successful Qt application that works together with a Java application server, one has to interface the two technologies in some way.

In large contrast to the Java application, these requirements are almost purely non-functional. The Qt application pretty much can gather all the functional requirements from the Java application. However, because the Qt application needs to interface with Java and since a proper graphing library for Qt is yet to be found, it would be pointless to start defining functional requirements before non-functional questions are answered.

6.5.2 High-Risk Elements

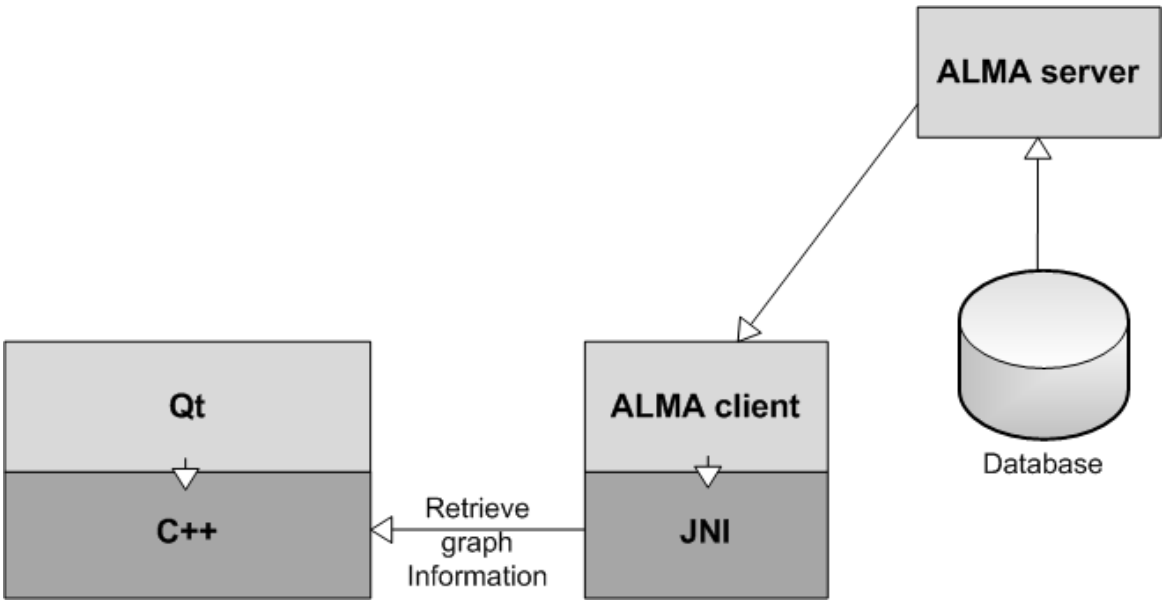
Concerning the requirements listed in the previous chapter, the only real question mark is to find a proper graphing library for Qt. Interfacing Qt (C++) with Java is something which will be built with ready-made tools which have been proven to work – so there's nothing there that could really go wrong. Of course implementing a proper interface to suit the needs of the project can prove to be difficult, but in this case it's something that can be worked out.

Other high-risk elements might rise during this iteration step, but mainly the high-risk elements are going to become more detailed and clear as more research is done.

6.5.2.1 Interfacing C++ with Java

I started the work by creating a small demo application to interface Qt (C++) with Java. This interface, as will the final interface when used with the ALMA system, creates a new Java Virtual Machine (JVM) using the ready-made Java Native Interface (JNI) libraries provided by Oracle, tailored specifically for these kinds of purposes. So, the Qt application launches a new JVM and within that JVM the usual ALMA client, which communicates with the ALMA server, is opened.

The next graph depicts how the interface from Qt to ALMA server is supposed to work in its entirety. Qt works on top of C++. On top of Qt/C++, there might be a third party, external, graphing library (not depicted in the graph). C++ uses, through JNI, an ALMA client, which communicates with the ALMA server. Everything that is needed to make the whole application work is included in the picture.



Graph 25: Interfacing a Qt application with ALMA server

As it turned out, interfacing from C++ to Java is much easier than the other way around because the C++ application can easily "drive" the Java application running within the JVM that it created. Creating new Java objects and calling Java methods from C++ is pretty straightforward and the return types that it can accept from methods can be of primitive types (such as Integers or Strings). If you wanted to use an object as a return type, which is essentially a composite of primitive types, you would have to go through more trouble—especially if that object has an inheritance hierarchy and composite members.

6.5.2.2 Interfacing with the ALMA server

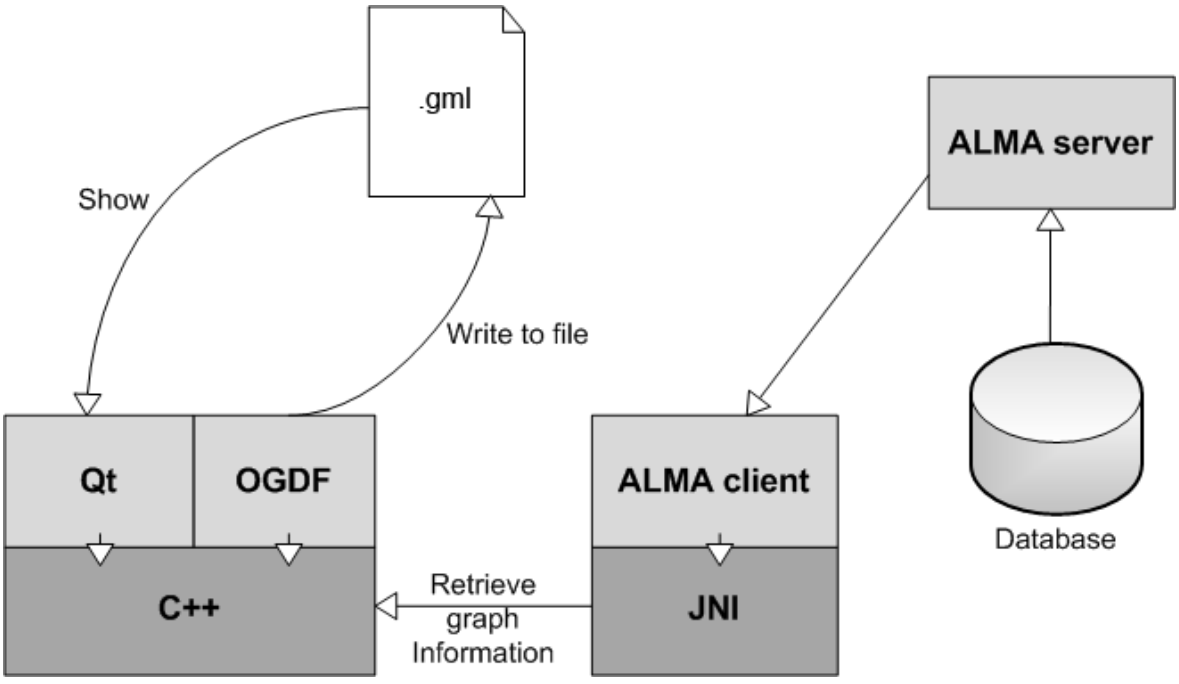
After successfully implementing a C++ to Java interface I wanted to expand on it by creating an interface to the ALMA server application. ALMA has in it already a remote client, which can be used in this case. In the Java application, running in the JVM, created by the C++ application, I started a new instance of an ALMA remote client. This step was really simple. All you actually have to consider is to include all the proper ALMA modules in your Java application. Starting the application is like starting any other Java application.

6.5.2.3 Graphing libraries for Qt

The riskiest part of the Qt application also proved to be fatal. After spending considerable time investigating on various different graphing libraries I came to the conclusion that while C++ might provide a working solution for this, Qt in itself doesn't have this kind of a tool in it yet. This essentially rendered the creation of the Qt application void—implementing a graphing library from scratch is too much work concerning the scope of this thesis and using a C++ graphing library wouldn't be much of a Qt application.

I went through a number of different graphing tools for C++, hoping that I could only implement the graphics through Qt somehow; the graph calculations could have been performed purely with C++. The most considerable options were BGL (Boost Graph Library) and OGDF (Open Graph Drawing Framework). However, concerning the quality of C++ graphing libraries it would have taken a lot of effort to make this happen. I got as far as creating graphs with C++ but I never truly found a good way to import the graphics of the graph to Qt. If I had managed that, then it would have provided a suitable solution.

The graph below illustrates how I planned to use OGDF with Qt. OGDF can create a graph and layout it using various layouting strategies. In this plan, OGDF creates the graph, layouts it and saves the graph in GraphML (graph markup language) format. Qt reads the created GraphML file and creates a view based on information found from the file.



Graph 26: Using an external graphing library in conjunction with Qt

I could have also reworked this thesis to concern only C++, instead of Qt, and compared it against Java. However, to implement all the various kinds of interactions, all the functional requirements, would have been tedious to work out using C++, not to mention creating all necessary UI components to work with.

Everything else panned out rather nicely when working with Qt. There were a number of difficult obstacles to overcome, but finally there was nothing to be done to bring the project all the way home.

6.6 Iterations for the Java application

Now that the core application structure is thought out, the rest of the requirements need to be taken care of. This namely means that plotting strategies should be able to define their own commands – the actions that the user executes when inspecting a network view differ from those of when he's inspecting a schematic. Some wrapping up is done as well to bundle all the different parts together. It's sort of like trying to standardize how the tool works or creating a façade to cover up some of the details that are not really necessary for the user to see.

Table 4: Java Iterations phase backlog

Priority	Requirement
1.	Create a plotter base class and a plotter for the Network views
2.	Create a plotter for the Schematic views
3.	Implement Commands
4.	Implement Memento
5.	Compose a façade for the whole system
6.	Construct a mechanism for choosing a layout algorithm
7.	Store/restore the state of a graph view

6.7 Iteration 1: Plotter Base Class and Network plotter

The difficulty with creating a base class for all the plotters was that beforehand you would need to figure out all the generic elements that you could sink in it. This was largely a matter of reading the yFiles documentation and getting to proper grips with it.

The plotter base class should house in it all the things that all the plotters require. Defining all these things separately in different plotters would create code that is hard to maintain. What makes creating the plotter base class easier is how the core model works – I could focus simply on how to accommodate all the things in yFiles, I didn't have to worry about ALMA centric things at all because the core model already takes care of all this.

The things you need to take into account with yFiles are largely things that concern layouting. I also decided to house things related to tabular views in the base class, though each plotter that wishes to utilize these items is responsible for maintaining them.

Concerning the network plotter there was really little that needed to be accounted for. The network plotter is a simple plotter and it sits on top of the plotter base class nicely. The network plotter simply plots everything concerning the objects you supply it with. The one thing that it does need to account for are user defined settings about how given things should be grouped and abstracted.

6.8 Iteration 2: Schematic plotter

The schematic plotter is a lot different from the network plotter, even though with enough levels of abstractions you could make the two as one. This was something however which seemed to require a lot of work so I decided to create a separate plotter for it altogether.

The schematic plotter is interested how things connect to each other. It's titled schematic plotter because most often it realizes schematic connections. You could of course use it for other purposes, to realize connections of other types, but at this time it felt like there would be little other use for it. Hence I declared the plotter as a schematic plotter instead of trying to come up with a more general description, meaning and name for it.

The difficulty with realizing the schematic plotter proved to be working with the yFiles table model. The table model in the yFiles deals with a lot of details that you

have to take into account. If you fail to implement some small detail you could be dead in the water without a proper error message dictating why that is.

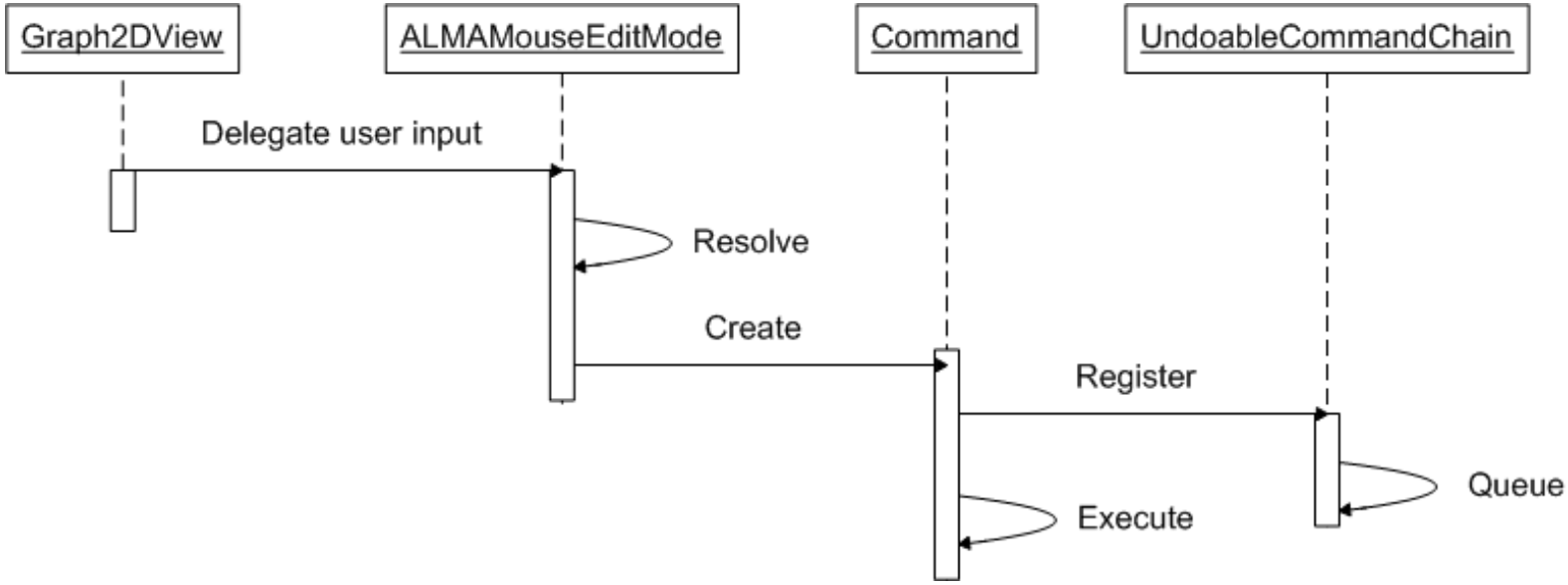
Also, creating a layouter for the schematic proved to be difficult because you need to explicitly instruct the layout algorithm to take certain things, like groups and tabular structures, into consideration. Incorporating groups into the schematic layout also required the generation of some amount of custom code so that the layout algorithm would properly calculate the size of the groups. All in all, creating the schematic plotter took a lot more time than what I had allocated for it.

6.9 Iteration 3: Commands

A plotting strategy creates the nodes and edges to the graph based on the rules built into it. Different plotting strategies have different kinds of actions that they associate with newly created nodes and edges. E.g. what should happen when a user clicks on a node? With a network strategy that means to expand the clicked node, revealing its hierarchy, link and type structure. Contrasted with a schematic strategy that means to deabstract it, i.e. to show its underlying elements.

For each plotting strategy a command factory can be assigned to it. This command factory takes as parameter the node or edge created and the factory will then return a list of commands that should be associated with it. This list of actions is linked to a customized EditMode which manages the clicking of nodes and edges. An EditMode is a construction in yFiles, which manages users interaction within the frame where the graph is placed.

If a plotting strategy does not define a command factory a default command factory is used. This default command factory is provided by the base class AbstractPlotter, which is the base class for all plotting strategies. The following sequence diagram illustrates how exactly the command structure works.



Graph 27: Command sequence diagram

6.9.1 Undoable Commands

Applications, which support undoability, offer the user a sense of security since they can always go back if they didn't mean to do something. To support undoability it takes a few tricks to make it happen and these tricks are command specific – you need to exactly define how to go back that one step. A few commands support undoability and this is of course something that can be extended in the future as well.

An undoable command is a normal command except that it implements UndoableCommand interface. This interface defines a method (not surprisingly) called undo. It is up to the command to register itself with an UndoableCommandChain.

The UndoableCommandChain keeps a stack of executed actions and when called upon it takes the command, which was last executed and calls the undo method on it. It then removes this command from the stack and moves it to another stack. This other stack exists so that the user can redo the command if he so chooses.

6.9.2 Effects

Effects are commands, which create an effect on the graph, which should then be cleaned up afterwards. An effect-command implements the Effect interface, which defines a method called dispose. An effect-command works with an instance of AppliedEffects.

The AppliedEffects instance is responsible for tracking all applied effects and on occasion calling upon the commands dispose method. It does this by starting a new thread, so that it doesn’t mess with the execution of the main program. If the dispose method returns true it means that the effect was removed from the graph and the AppliedEffects instance then removes this effect from its list of effects.

An example of this is the local-edge command. The local-edge command highlights all the arriving and leaving edges of a node and it works in the following way:

1. The user activates a node to display all incoming and outgoing edges by executing the local-edge command.
2. The local-edge command highlights the edges and registers itself with an AppliedEffects instance.
3. Every time the dispose method is called by the AppliedEffects instance, the local-edge command executes a series of steps to figure out if the effect should be removed. The local-edge command does this by checking whether the user still has focus on the node he clicked. If the user has moved the cursor away from the node, the command will then remove the applied effect from the view.

6.10 Iteration 2: Saving the program's running state

The Memento Pattern is used in two places. First it's used with ALMAGraph2D to store its state. This stored state is used for example by an undo command to fall back when a user decides to undo his action.

Second, and much more importantly, the Memento Pattern is used to save the current state of the graph being inspected, i.e. all the defining characteristics when viewing a graph: roots of the graph, layout algorithm, plotter, scope, include/exclude types, etc. This mechanism is also used when the user customizes how he would like the graphing solution to work since this Memento object can restore the graph from a text representation. So the user writes (or uses a separate application) to create a text representation of the Memento object. He then loads the text representation to the Memento object and in return he gets a ready-made graph.

To support the use of a Memento object, all the objects in it need to be able to be represented in text format. Rather than creating a serialized string of them, it would be better to be able to associate an object type with an ID. This ID could

then be used by the application to create an object of this type. The way this works is nothing too exciting, just associate an ID with an object type and then create new object based on that type.

Java enumerated types offer quite a lot of functionality if contrasted with enumerated types in C++. In Java, enumerated types are more like special cases of classes. In fact when you compile an enumerated type in Java, the compiler turns it into a regular Java class. (Niemeyer and Knudsen 2005, 150-153)

The point of all this is that when you're creating enumerated types in Java you can explicitly declare values corresponding to different types. I.e. you can associate an enumerated type with a static ID and that ID could be anything you like (Niemeyer and Knudsen 2005, 150-153). In case of ID's it's usually best to declare them as Integers. This provides a perfect solution for me to associate a type with an ID – enumerated types associated with an ID.

See section 4.6.8 for a detailed description of the Memento Design Pattern.

6.11 Iteration 3: Façade for GraphConfiguration

The Façade object, which is titled GraphConfiguration, is a concept, which is created just for convenience sake. The GraphConfiguration houses within it all the major parts of the graph. The major parts of the graph include:

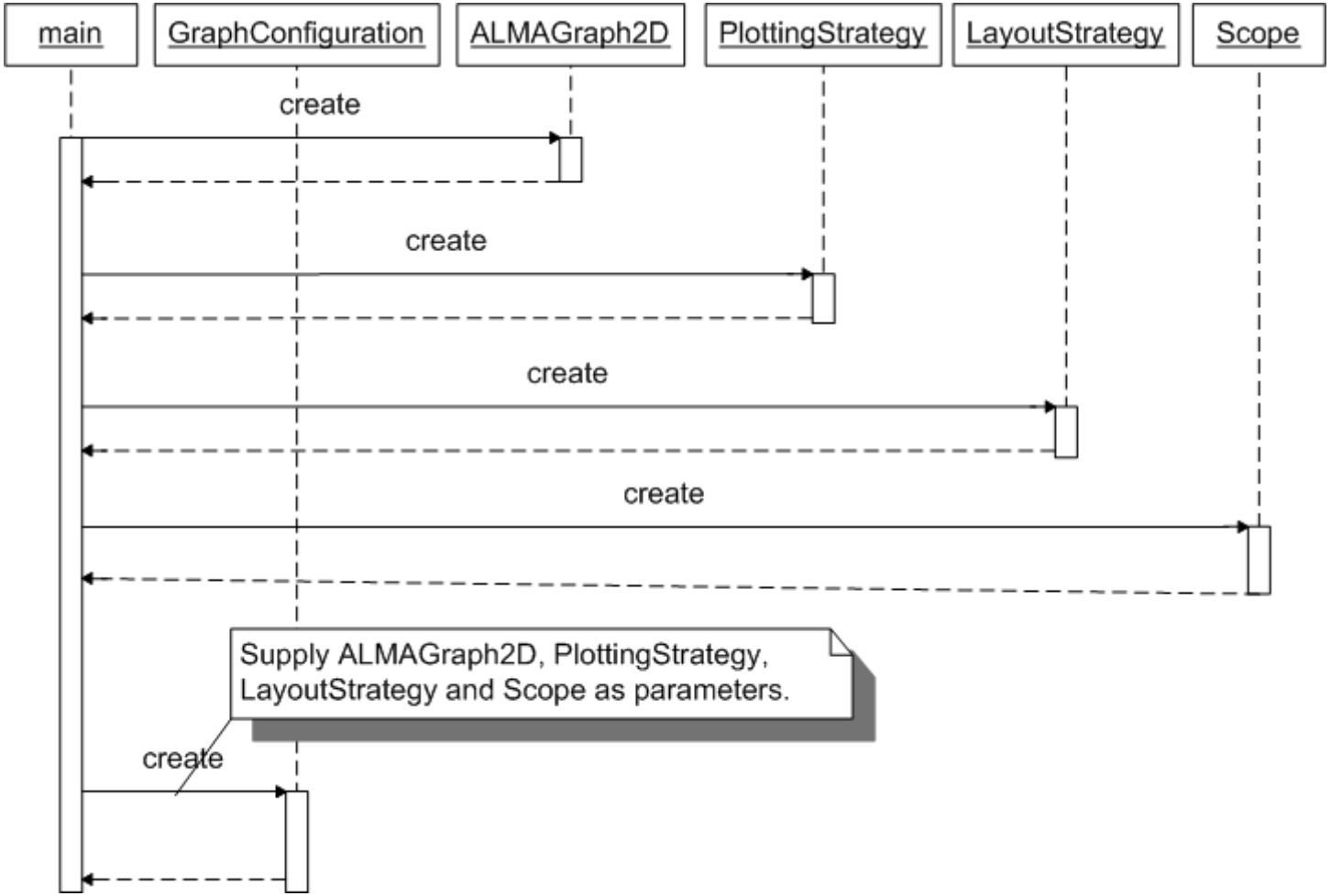
1. An ALMAGraph2D instance
2. The plotter
3. The layout algorithm
4. The scope
5. Display mask
6. Exception maps to exclude certain types of elements from the graph.
7. Memento object to save/restore the state of the current view
8. Preferred window height and width
9. Selection associations
10. Grouping associations

11. Abstractable elements
12. Interactive abstractable elements.

In addition to providing a central point of access to all the major parts of graph creation and manipulation, the GraphConfiguration instance is the sole element, which is used when the user initiates the creation of a new graph. For example, to create a new graph view, the user creates a new GraphConfiguration object and specifies for it the following items:

1. The graph to use (i.e. which roots to use)
2. The plotter to use (Network, Schematic, etc.)
3. The scope
4. And the layout algorithm

Additionally the user can also define which things he would like to exclude from the graph and so on, but to get started you'll only need four things – four simple questions.



Graph 28: Creating a new GraphConfiguration object.

The GraphConfiguration, in essence and among other things, realizes the Façade Pattern. You can find a description of the Façade Design Pattern from chapter 4.6.9

6.12 Iteration 4: Choosing a layout algorithm

The layout algorithm isn't chosen by the program, the user predefines it. Of course you can't really use a schematic layout algorithm with a network plotter but you could choose to use a balloon layout or a hierarchical layout algorithm with the network plotter, etc.

The chosen layout algorithm works with the assigned plotter. It takes the ALMAGraph2D instance created by the plotter and tries to layout it. Implementing most of the layout algorithms is a simple enough task. Though with tabular structures (with schematics) you need to take care of various sorts of things.

At first, the Visitor Pattern was used to associate a layout algorithm with a plotting strategy. However, I decided to drop the Visitor Pattern from the final design because it offered very little but added some complexity to the final design. Also, worth of note is that the Visitor Pattern is hardly intended for this kind of purpose.

6.13 Beta Tests

A rigorous testing period ensued after the last iteration ended. Lucky for me, I could use a test database with loads of information in it. Most of the time I was testing and configuring schematic views. Network stuff was really much simpler and it seemed to flow on it’s own (nothing interesting in it really).

It quickly became evident that although the schematic views are able to produce graphs with tens of thousands of elements (nodes and edges) it would soon grow

pretty slow to handle these amounts of information. The strain is not entirely on the computer; it's hard for a human being to take in that much information on one go as well. This goes to show that there is room for improvement and optimization. For now, things work out well enough though.

6.14 Deployment

Deploying the application consists of the following tasks:

1. Include the application in the main build of the whole ALMA system.
2. Design and implement point(s) of entry for the application
3. In a license file include the graph tool, if the customer has bought a license for it.

These steps were trivial to carry out.

7 RESULTS

No piece of software is ever ready, they say. If an application doesn't evolve it dies, they say. However, this development cycle has come to an end. The following is a brief summary of the work, performance, the design process, a personal reflection on software engineering and a brief view on Qt vs. Java.

7.1 Summary of the work

The Java application works, "adequately" as you can read from the following chapter. The Qt application never survived past the Elaboration phase. In fact, it should have never survived past the Inception phase. However, this was a small setback concerning that the time allocated for the Qt application would have been spent in any case — where exactly it was spent is just a matter of definition.

While not being able to "drive home" the Qt application it was still fun to work with the collateral techniques involved with it. Marrying a C++ application with a Java application, though it proved to be pointless, was much fun. Seeing two disparate technologies work together was educating and almost even exciting. Trying out the various ways in which to do Qt and C++ development was also, pointless, but enlightening.

To list the troubles concerning the Java application the biggest setback was underestimating the required time concerning implementing schematic layouts. The schematic layouts require a tabular structure and, though the yFiles library does support tables, it was still really difficult to realize and implement them. Besides this hindrance Java development went along pretty well.

A satisfactory end result was produced which manages to fulfil the primary goals set for the project by the people who initially expressed the interest to develop the software. Despite the project not being able to deliver a working solution for Ot, I myself am pleased with the outcome.

7.2 Performance—heuristic

It's difficult to give a single answer about how well the application performs. It, of course, depends upon hardware but it also depends on structure complexity (i.e. graph structure complexity). Also, schematic layouts, which employ tabular structures, add their own weight not to mention network capabilities (because the application works as a client-server system), which can vary radically between locations.

There are four major aspects related to forming graphs. First one is to query ALMA server for information. Second, based on information received from the server a small part of the graph is plotted. This second part is reiterated until some condition is met and the graph is considered to be ready. The third part is to feed the whole plotted graph to a layouter provided by the yFiles library. The fourth aspect is rendering the graph to the screen and navigating in it.

It is, of course, impossible to optimize the third party library used for layouting the graphs, i.e. yFiles, and thus yFiles sets the par in some manner. But it sets the par so far that it's practically impossible to reach it. For the same reasons it is also impossible to affect the performance of rendering the graph to screen. There is much room for optimization in the manner which the application traverses the ALMA system structure. Currently a number of tricks—tricks that hinder the application while making it easier to work with—are employed when working with the structures of the ALMA system. These tricks could be worked into more robust solutions and thus enhance the performance.

Varying upon layouting algorithm used, the system can realize graphs with 1,000 - 10,000 objects. Tens of thousands of objects start to hinder execution and result in long wait times and great difficulty in navigating the graph. While it would be nice to optimize the system to realize hundreds of thousands of objects, how could any user read the resulting graph? It's already pretty difficult to read graphs with more than a hundred objects. Based on this reasoning I rated the performance as "adequate". When inspecting larger graphs, it is wise to narrow the view down by some limiting mechanism.

There is, however, one way to look at huge graphs (more than 10,000 objects) in a way that it makes sense and that one way is tabularized views or some other means of grouping objects. It's not pretty, nor easy, to inspect a graph with close to a thousand objects in it but on occasion it can be enlightening to see the bigger picture even though one does not follow every detail in it.

In conclusion, optimizing the process would yield benefits but at this point these benefits are still a question mark. This need is largely a question of whether a real-world-user somewhere, sometime requests it. So, based on a "gut-feeling" (you might also call it a guess) the performance is "adequate".

7.3 Performance—benchmarking

Even though little—or none at all—is to be gained with benchmarking, I thought it would be at least, if nothing else, fun to do it. Besides one has to satisfy ones geeky nature.

This benchmarking session is hardly scientific-grade but hopefully it yields results that are more useful than just fun statistics. Hopefully they will be at least somewhat informative. Following aspects are taken into account when benchmarking:

- Number of nodes
- Number of edges
- Layout algorithm used
- Java heap space

These points are rather self-explanatory. Recording different Java heap spaces gives some insight into the minimum requirements of the application and provide insight of the consequences of having smaller amounts of memory.

Three separate graph-forming aspects are recorded in this benchmark: graph creation, graph layouting and rendering a graph to screen. Two types of graphs

are used in this benchmark session: cyclic graphs and connected acyclic graphs (i.e. trees). Three kinds of layouters are used: hierarchic, organic and circular.

The benchmark results were recorded with the following system:

- Intel(R) Core(TM) 2 CPU T7200 (each core clocks at 2.0 GHz)
- 3.00 GB (RAM) memory
- Windows Vista 32 bit.

To note: the computer performing the benchmarks is not optimized for benchmarking in any manner and there could very well be undesired background processes running wild. Like said before, this is hardly a scientific-grade benchmark session, the only purpose is to produce at least some kind of information.

All the results were taken from a similar set of objects, i.e. the structure of the graph does not introduce unstable elements regarding benchmarking. All the graphs realized need to be connected and since it's impossible to create a tree of e.g. 100 nodes and 10 edges, because the number of edges in a tree is exactly one less than the number of nodes—hence the lack of results for all permutations.

Each permutation is ran 10 times and the average of these values is recorded, while excluding radically differing results. Radically differing results are those that differ by 10% from the average.

There exist a few known reasons for potential differing results. One is something that will happen each time a benchmark session is executed. Java uses Just-In-Time (JIT) compilation when it executes byte code (i.e. a program). A JIT compiler compiles the structures it needs just before execution (just like it's name cleverly states). This means that the first pass of a benchmark is bound to encounter hindrance. This is however easy to filter out from the results.

Second potential reason for differing results is that the computer performs some arbitrary operating system routines in the background, which reduce the amount of available clock cycles.

The notation *P* stands for the plotter used (hierarchic or network) and *H* marks the allocated heap space for the JVM (24 MB, 128 MB and 1024 MB—all amounts are in mega bytes unless otherwise stated). L stands for layout algorithm. All recorded times are in milliseconds, notated as *[ms]* in the table. If some cell in the time column is annotated with *N/A* it means that it is not possible to plot that graph because there is not enough memory available.

The following benchmark results (again, permutations of number of edges, number of nodes, heap space and plotter/layouter used) were recorded:

Table 5: Results for plotting Hierarchic and Networked graphs

P	H	Nodes	Edges	Time [ms]
Hierarchic	24	10	9	63.00
		100	99	343.90
		1,000	999	3,296.89
		10,000	9,999	N/A
	128	10	9	94.67
		100	99	511.22
		1,000	999	2,176.22
		10,000	9,999	26,833.22
	1024	10	9	65.89
		100	99	328.63
		1,000	999	2,946.78
		10,000	9,999	31,579.22
Network	24	100	999	304.22
		1,000	9,999	N/A
	128	100	999	264.00
		1,000	9,999	5,564.11
	1024	100	999	334.67
		1,000	9,999	6,654.44

Table 6: Results for using Hierarchic layouter

L	H	Nodes	Edges	Time [ms]
Hierarchic	24	10	9	8.00
		100	99	19.44
		100	999	30,045.56
		1,000	999	460.38
		1,000	9,999	N/A
		10,000	9,999	N/A
	128	10	9	9.22
		100	99	25.33
		100	999	19,846.89
		1,000	999	276.33
		1,000	9,999	x
		10,000	9,999	34,268.56
	1024	10	9	7.78
		100	99	19.22
		100	999	18,985.56
		1,000	999	258.78
		1,000	9,999	1,209,462.11
		10,000	9,999	32,581.89

Table 7: Results for using Organic layouter

L	H	Nodes	Edges	Time [ms]
Organic	24	10	9	10.22
		100	99	208.60
		100	999	1,479.00
		1,000	999	14,721.44
		1,000	9,999	N/A
		10,000	9,999	N/A
	128	10	9	11.89
		100	99	215.89
		100	999	1,460.67
		1,000	999	14,560.78
		1,000	9,999	15,953.33
		10,000	9,999	48,604.44
	1024	10	9	10.00
		100	99	214.00
		100	999	1,420.89
		1,000	9,999	14,600.90
		1,000	9,999	15,005.33
		10,000	9,999	45,965.22

Table 8: Results for using Circular layouter

L	H	Nodes	Edges	Time [ms]
Circular	24	10	9	5.78
		100	99	12.44
		100	999	104.00
		1,000	999	362.33
		1,000	9,999	N/A
		10,000	9,999	N/A
	128	10	9	9.78
		100	99	11.11
		100	999	27.22
		1,000	999	135.56
		1,000	9,999	325.11
		10,000	9,999	4,924.11
	1024	10	9	6.22
		100	99	12.11
		100	999	21.00
		1,000	999	101.89
		1,000	9,999	282.11
		10,000	9,999	3,041.44

Table 9: Results for rendering a graph on the screen

H	Nodes	Edges	Time [ms]
24	10	9	848.33
	100	99	834.89
	100	999	880.78
	1,000	999	1,340.00
	1,000	9,999	N/A
	10,000	9,999	N/A
128	10	9	838.78
	100	99	836.78
	100	999	x
	1,000	999	822.78
	1,000	9,999	x
	10,000	9,999	8,957.44
1024	10	9	849.00
	100	99	844.00
	100	999	848.89
	1,000	999	869.89
	1,000	9,999	1,174.56
	10,000	9,999	1,373.56

The benchmarked results don't tell us very much. What is of interest are the plotter results (hierarchic and network) because they are what this thesis was about. The results display that if the number of elements in a graph grows by a factor of 10, the execution time also increases roughly by the same factor. Linear efficiency is good enough.

Nothing too exciting can be found from different layouter results. They tell that some layouters are faster than others. Circular layouter is by far the fastest one. Organic and hierarchic are about the same.

Some aspects affect different layout algorithms profoundly. For example, the number of nodes does not affect as much as the number of edges. Every time the layout algorithm needs to account for an edge, it needs to calculate that it doesn't overlap existing edges or nodes—an operation, surely, with rapid increase in time complexity as the number of edges gets bigger. This aspect has radically different weights on different layout algorithms, as can be seen from the results, causing peculiar anomalies to surface. One of these anomalies is the hierarchic layouter when presented with 1,000 nodes and 10,000 edges—time complexity increases by almost a factor of 10,000 while the number of items increases by a factor of 10, rendering the layouter totally useless in production grade software. But, to be fair, this sort of a graph is hardly ever to exist "in the wild". It does, however, indicate a potential weakness in the system. How exactly that weakness might manifest itself in a production system doesn't really matter—the scenario is far fetched to say the least.

Rendering the graph on the screen is pretty much a constant—it doesn't matter how much stuff the graph contains.

It is interesting to note that different allocated heap sizes don't affect the performance all that much. Some of the results indicate that it's a factor but not in a significant magnitude.

Java uses a system called garbage collection to clean up things from the memory it no longer needs. If contrasted with for example C++, there is no such mechanism, which means that the programmer is responsible for managing memory. In Java a

garbage collector routine is executed every now and then to release memory that is no longer used. What I was hoping to see in these benchmarks would have been the impact of automated garbage collector when working with limited amounts of memory. In theory, when memory runs low, the garbage collector is ran rather frequently which would hinder the whole system down by some unknown factor—garbage collection being a rather heavy operation. Consider the following graph, which compares garbage collector activity between a system, which has 24 MB of heap, and a system that has 1024 MB of heap at its disposal. Both JVMs are executing the same program. The top image is the system with 1024 MB heap; the bottom image is the system with 24 MB heap. The y-axis marks the activity of CPU and garbage collector; the x-axis is time. The blue lines correspond to garbage collector activity while the yellow lines mark the overall CPU usage.

Graph 29: Garbage collector activity. Top image: 1024 MB heap space; bottom image: 24 MB heap space. Blue = garbage collector activity. Yellow = overall CPU activity.

At some point, in a system with a limited amount of heap space, the garbage collector is ran quite frequently and it accounts for a lot of CPU usage. This scenario is, however, hard to come by. It takes some thinking to even come up with a benchmark scenario that renders these kinds of results. So, in short, the garbage collector is hardly to cause any significant performance loss.

That's about all that could be learned from these benchmarks.

7.4 Evaluation of the design process

The design process was flawed from the very beginning. The main reason for the partially failed end result was to neglect deep enough research before adhering to work on the problem. I'm referring to the fact that I didn't research the possibilities of Qt to work with graph structures enough. It was a major error in judgment to leave something to faith and pursue Java technology before understanding and solving all the problems with both technologies.

Though I don't put much weight on following practice it's true that this error in judgment was primarily because a major UP practice was ignored—i.e. to tackle high-risk and high-value issues early in the inception phase of the project. Failure to adhere to this practice caused one of the high value and high-risk problems to be put on hold until inferior problems were solved.

Also, as nice as the idea of writing unit tests before you write any code sounds, my personal view is that it is increasingly difficult to maintain tests when reworking the core application logic frequently. When you rewrite your core application logic (which turned out to happen all the time in my case) you also have to redesign, rewrite and reimplement all your test cases. At times I found myself throwing away unit tests because they no longer served any purpose. This approach could be feasible when used in a surrounding where one is more comfortable with the ins and outs of the core problems or when a person would have a higher degree of professional experience with TDD (Test Driven Development).

7.5 On the issue of Java vs. Qt

This modest (and largely biased towards Java) study, which tried to compare Java and Qt UI capabilities, fell short on its tracks. However something can be deduced from the fact that using Java technologies this visual representation of the ALMA system was doable whereas with Qt I was unable to find a proper visual graphing library to do the job. When it comes to comparing Qt and Java one can say that they both have their pros and cons and it seems really pointless to try and force one over the other.

There's a vast amount of people supporting Java and I'm sure that there's also a vast amount of people supporting Qt. Most of the time, when someone promotes a technology, those opinions reflect person's own interests towards the technology. There really is no authority, which can say that Java is better than Qt or the other way around.

7.6 General Reflections on Software Engineering

In a word, software engineering is hard. It's not so much about tackling different kinds of techniques or methods; it's the whole process in general – the individual parts are easy, the whole is a handful. Software engineering entails working ideas, which form in your mind as perfect, into a functional piece of code. A piece of code that is responsible for accounting for all the ways in which your perfect idea fails. It's a big leap from idea to code, one that most often holds within it more than you originally planned for.

What makes things even more difficult is reading too much into "sound" advice. There is no right way to do software and making the mistake of following advice blindly will eventually lead you astray. This goes pretty much for every single book about software engineering. They have a habit of constraining your mind with rules without telling you that in 90% of cases, they will not work for you.

The most important thing I've learned about software engineering is that it's not about software—or engineering for that matter. What matters is the end product and if it serves to benefit the people who use it. Everything else is noise. Software engineering is a tool, one that is flawed in most ways.

REFERENCES:

yWorks. *About yFiles for Java.* 2010. http://www.yworks.com/en/products_yfiles_about.html (accessed November 22, 2010).

Yegge, Steve. *Get that job at Google.* 12 March 2008. http://steve-yegge.blogspot.com/2008/03/get-that-job-at-google.html (accessed December 12, 2010).

Vandenburg, Glenn. *The Real Software Engineering.* Directed by EdgeCase Software Artisans. Performed by Glenn Vandenburg. JRubyConf, Columbus. 2010.

Vesterholm, M., and J. Kyppö. *Java-Ohjelmointi.* 6th Edition. Kamppi, Helsinki: Talentum oyj, 2006.

ALMA Consulting Ltd. *ALMA - Transferring Knowhow.* 2010. http://www.alma.fi/In_english/Front_page (accessed September 15, 2010).

Apache Software Foundation. "Apache Ant User Manual." *Apache Ant.* 15 November 2010. http://ant.apache.org/manual/ (accessed November 15, 2010).

Black, Paul E. *acyclic graph.* 19 April 2004. http://xlinux.nist.gov/dads/HTML/acyclicgraph.html (accessed July 30, 2011).

—. *connected graph.* 19 April 2004. http://xlinux.nist.gov/dads/HTML/connectedGraph.html (accessed July 30, 2011).

—. *directed graph.* 20 November 2008. http://xlinux.nist.gov/dads/HTML/directedGraph.html (accessed July 30, 2011).

—. *forest.* 19 April 2004. http://xlinux.nist.gov/dads/HTML/forest.html (accessed July 30, 2011).

—. *in-order traversal.* 14 August 2008. http://xlinux.nist.gov/dads/HTML/inorderTraversal.html (accessed July 30, 2011).

—. *postorder traversal.* 14 August 2008. http://xlinux.nist.gov/dads/HTML/postorderTraversal.html (accessed July 30, 2011).

—. *preorder traversal.* 14 August 2008. http://xlinux.nist.gov/dads/HTML/preorderTraversal.html (accessed July 30, 2011).

—. *red-black tree.* 23 May 2011. http://xlinux.nist.gov/dads/HTML/redblack.html (accessed July 30, 2011).

—. *tree.* 14 August 2008. http://xlinux.nist.gov/dads/HTML/tree.html (accessed July 30, 2011).

CVS. "Introduction to CVS." *Introduction to CVS.* 22 November 2010. http://www.nongnu.org/cvs/ (accessed November 22, 2010).

fastutil. "Introduction." *Introduction.* 22 November 2010. http://fastutil.dsi.unimi.it/ (accessed November 22, 2010).

Freeman, E, E Freeman, B Bates, and K Sierra. *Head First Design Patterns.* Sebastobol, California: O'Reilly Media, Inc., 2004.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Massachusetts: Addison-Wesley Professional, 1994.

Grzegorz, Szewczyk. "Comment." *Comment.* April 2011.

Holowczak, Richard. *Programming Concepts.* 29 May 2007. http://cisnet.baruch.cuny.edu/holowczak/classes/programming/ (accessed January 12, 2011).

Knuth, Donald. *The Art of Computer Programming.* 3rd Edition. Vol. 1. 3 vols. Reading, Massachusetts: Addison-Wesley Professional, 1997.

Larman, C. *Agile and Iterative Development: A Manager's Guide.* Reading, Massachusetts: Addison-Wesley, 2003.

—. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* 3rd Edition. Upper Saddle River, New Jersey: Prentice Hall PTR, 2005.

Larman, C., and V. Basili. "Iterative and Incremental Development: A Brief History." *IEEE Computer*, 2003.

Niemeyer, P., and J. Knudsen. *Learning Java.* 3rd Edition. Sebastopol, California: O'Reilly Media, Inc., 2005.

Nokia. *What is Qt.* 2010. http://qt.nokia.com/ (accessed November 22, 2010).

Moock, Colin. *moock.org.* 21 January 2011. http://www.moock.org/lectures/mvc/.

Oracle. *About the Java Technology.* 2010. http://download.oracle.com/javase/tutorial/getStarted/intro/definition.html (accessed November 22, 2010).

—. *Java Native Interface: Programmer's Guide and Specification.* 2010. http://java.sun.com/docs/books/jni/ (accessed November 22, 2010).

Pressman, R. *Software Engineering: A Practitioner's Approach.* 4th Edition. New York, New York: McGraw-Hill, 1997.

Squidoo. *Arwen Undomiel.* 12 January 2011. http://www.squidoo.com/arwen-undomiel (accessed January 12, 2011).

APPENDIX: USE CASES

Use Case Section	Comment
Use Case Name	Initiation of the application
Preconditions	The ALMA client needs to be properly set up.
Main Success Scenario	Using the ALMA client, the user chooses an object he would like to inspect with the graphing application. The application plots the graph according to predefined settings.
Exceptions	The selected root does not produce any results leading to an empty graph. This might be the result of filters, which limit out the selected item and its connections.
Frequency of Occurrence	Continuous

Use Case Section	Comment
Use Case Name	Interacting with the application
Preconditions	None
Main Success Scenario	The user is able to interact with the graphing application so that he is able to inspect a given element more deeply. The element to inspect can be a node or an edge. For future reference, the user-supported actions should be something that can be easily expanded/subtracted later on. In other words this part of the application should be highly extendable and customizable. To start with, it's enough that the user is able to extend his selection to see more of the ALMA system structure.

Exceptions	None
Frequency of Occurrence	Continuous

Use Case Section	Comment
Use Case Name	Defining parameters
Preconditions	None
Main Success Scenario	The user is able to define what he wants the graphing application to show to him. The user is able to decide whether he wants to see the hierarchy, the links, the type structure or any permutation of the latter described list. The defined parameters are stored in the ALMA system in text format. To manipulate these configurations might prove to be arduous, so in the future a separate application for forming configuration files needs to be created.
Exceptions	None
Frequency of Occurrence	Seldom

Use Case Section	Comment
Use Case Name	Security
Preconditions	None
Main Success Scenario	For security reasons, the application should be identifiable by different kinds of users. This is something that is handled by the ALMA client but on some rare occasions issues might rise which need to be

	taken into consideration.
Frequency of Occurrence	Random
Miscellaneous	For future development

Use Case Section	Comment
Use Case Name	Extending schematic connections based on a pivot
Preconditions	None
Main Success Scenario	The user selects a point of origin, a pivot, and relating to that pivot the schematic connections to and from that object are mapped from the beginning until the end. This is a useful tool for a user who wants to inspect a single object in the system to quickly see where it connects to.
Frequency of Occurrence	Random
Miscellaneous	For future development

Use Case Section	Comment
Use Case Name	Inspecting electricity distribution
Preconditions	None
Main Success Scenario	The user wants to see how electricity is distributed in the facility. I.e. in the application settings everything else needs to be filtered out except electricity related items.

Frequency of Occurrence	Random
Miscellaneous	For future development