Faculdade de Engenharia da Universidade do Porto

A Control for Graph Representation and Interaction

Tiago Cunha

Report of Project
Master in Informatics and Computing Engineering

Supervisor: António Fernando Vasconcelos Cunha Castro Coelho (Eng.)

June 2008

© Tiago Cunha, 2008

A Control for Graph Representation and Interaction

Tiago Cunha

Report of Project

Master in Informatics and Computing Engineering

Approved in public presentation by the jury:

Chair: Rosaldo José Fernandes Rossetti (Eng.)

____

Main Examiner: José Carlos Ramalho (Eng.)

Examiner: António Fernando Vasconcelos Cunha Castro Coelho (Eng.)

31st July, 2008

Resumo

As tecnologias de visualização de informação tornaram-se, nos últimos anos, ferramentas indispensáveis para a compreensão de conjuntos complexos de informação. Contudo, o desenvolvimento de aplicações neste campo foca-se muitas vezes em domínios específicos que não permitem generalização. Grafos são estruturas matemáticas, que podem ser usados para modelar relações entre objectos que representem conceitos, e, por conseguinte, ser utilizados para generalizar a representação de redes de informação.

Este relatório apresenta o *Olympus Graph Editor*: um controlo e um conjunto de bibliotecas que utiliza grafos como uma forma de representar e interagir visualmente com com redes de informação. Este controlo permite a criação, edição e interacção com grafos de forma rápida e prática.

Foi feita uma extensa revisão do estado da arte sobre teoria de grafos, técnicas de representação, tesauros, organogramas e ferramentas gráficas já existentes. Foi depois definida uma metodologia baseada no paradigma de arquitectura MVC com a intenção de manter as bibliotecas do controlo modulares e escaláveis para outros projectos. Foi pensado um paradigma de interacção para providenciar uma interface agradável ao utilizador, e foram implementadas várias técnicas de apresentação para representar a informação de uma forma esteticamente agradável.

Embora o controlo tenha sido desenhado para representação de informação genérica com grafos, foi feito um trabalho adicional para representar os conceitos presentes um sistema de gestão de arquivos: tesauros e diagramas organizacionais (*organogramas*). Finalmente, para avaliar e demonstrar as funcionalidades do controlo, é ainda implementada e apresentada uma aplicação protótipo.

Abstract

Information visualization technologies have become indispensable tools for making sense of complex sets of data over the last few years. However, the development of applications in this field often focuses on specific domains that do not allow generalization. Graphs are mathematical structures, that can be used to model relations between objects representing various concepts and,hence, be used to generalize the representation of information networks.

This paper presents the *Olympus Graph Editor*: a control and a set of libraries that uses graphs as a means to represent and interact visually with information networks. The control allows the creation, edition and interaction with graphs in a fast and practical form.

Extensive literature review on graph theory, graph representation techniques, thesauri, organograms and existing graphical tools was undertaken. A methodology based on the MVC architectural paradigm was then defined, with the intention of keeping the component libraries modular and scalable for other projects. An interaction paradigm was thought out to provide a user-friendly interface, and several layout techniques were implemented to represent information in an aesthetically pleasant way.

Although the control is designed for generic information representation with graphs, additional work was made in terms of representing concepts present in an archive management system: thesauri and organization charts (*organograms*). Finally, to evaluate and showcase the control's functionality, a prototype application is also implemented and presented.

Acknowledgments

I'd like to express my acknowledgments to all those who have, in some way, contributed to the realization of this project:

- To ParadigmaXis S.A, in the person of its CEO Eng. Alexandre Sousa for the opportunity of working in this company during the twenty weeks of the project.
- To Engª. Fernanda Ribeiro for the insight given into thesauri representation and graph drawing as a means to represent thesauri.
- To FEUP and to the MIEIC course, in the person of my project supervisor Eng. António Coelho for all the information, guidelines and opinions given with the intention to make this project the best it could be.
- To my project coordinator in ParadigmaXis, in the person of Eng. Filipe Correia for the countless hours spent discussing and tossing ideas around, and for the effort put into making ParadigmaXis my second home during the project.
- To fellow colleagues at ParadigmaXis for sharing their work place and their smiles with me.
- To my friends Ricardo Nuno Almeida, Marília Goncalves and Filipe Lemos for the lunches, the patience, the chats and the help during these last months.
- To my parents and brothers for the support and words of encouragement even when I got home late for dinner.
- To my girlfriend Silvia Silva, for being there always, and for not getting angry at me for all the extra time I spent working and could have spent with her instead.
- And last, but not least, to my father, for being my source of inspiration through all of my academic years. This work is dedicated to you, may I one day achieve half as much as you.

Tiago Cunha

"The most overlooked advantage to owning a computer is that if they foul up there's no law against wacking them around a little."

Joe Martin

Contents

1 Introduction **1**
1.1 Project Context . . . 1
1.2 Project Description . . . 2
1.3 Motivation and Objectives . . . 2
1.4 Paper's Structure . . . 3

2 State of the Art **5**
2.1 Graph Theory . . . 5
2.2 Graph Drawing . . . 8
2.3 Thesaurus and Organograms . . . 11
2.4 Data Structures . . . 13
2.4.1 Adjacency List . . . 13
2.4.2 Adjacency Matrix . . . 14
2.5 Graph Layout Algorithms . . . 15
2.5.1 Force Directed Spring Model . . . 17
2.5.2 Kamada-Kawai . . . 18
2.5.3 Fruchterman-Reingold . . . 19
2.5.4 Sugiyama-Misue . . . 22
2.5.5 Simmulated Annealing . . . 24
2.6 Graphic API . . . 26
2.6.1 High Level APIs . . . 26
2.6.1.1 Piccolo . . . 26
2.6.1.2 Netron . . . 28
2.6.1.3 JUNG . . . 28
2.6.1.4 Prefuse . . . 29
2.6.2 Low Level APIs . . . 30
2.6.2.1 System.Drawing . . . 30
2.6.2.2 Cairo . . . 31
2.6.2.3 GraphViz . . . 32
2.7 Graph Interaction . . . 33
2.7.1 Graph Oriented Interfaces . . . 34
2.7.1.1 MusicMap . . . 34
2.7.1.2 The Opte Project . . . 35
2.7.1.3 Websites as Graphs . . . 35
2.7.1.4 TouchGraph . . . 36
2.7.1.5 Visual Thesaurus . . . 37
2.7.2 Visualization Tools . . . 38

2.7.2.1 We Feel Fine 38
2.7.2.2 Web Trend Map 2008 39

3 Methodology **41**
3.1 Problem Description and Objectives 41
3.2 Use Cases 44
3.3 Solution Architecture 46
3.3.1 Data Models Discussion 46
3.3.2 Layout Algorithms Discussion 49
3.3.3 Graphic Library Choice 51
3.3.4 User Interface Principles 55
3.3.4.1 Design Principles 55
3.3.4.2 Interaction Paradigm 56
3.3.5 Architecture Description 58

4 Implementation **63**
4.1 The Calliope Library 64
4.1.1 The Node Class 65
4.1.2 The Edge Class 66
4.1.3 The Graph Class 67
4.2 The Athena Library 69
4.2.1 The Shape Class 69
4.2.2 The Line Class 71
4.3 The Kratos Library 72
4.3.1 The Stylesheet Class 73
4.3.2 The GraphPanel Class 74
4.4 The GIC Prototype 78
4.5 Final Remarks 82

5 Conclusions and Future Work **85**
5.1 Work Developed 85
5.2 Achieved Objectives 86
5.3 Future Work 88

References **94**

A Types of Diagrams in ISO 2778 **95**

B Class Diagrams **97**

C Layout Algorithms in the GIC **99**

List of Figures

2.1 The Bridges of Königsberg problem. . . . 6
2.2 A simple example of a graph. . . . 6
2.3 A sparse graph (left) and a dense graph (right). . . . 8
2.4 Types of graph drawing. . . . 10
2.5 Different graph conventions applied to the same graph. . . . 11
2.6 A graph and it's corresponding adjacency list. . . . 14
2.7 A graph and it's corresponding adjacency matrix. . . . 15
2.8 Different aesthetic criteria result in different graphs. . . . 16
2.9 Frame limiting in FR algorithm. . . . 21
2.10 The Piccolo object structure. . . . 27
2.11 Graph types in the JUNG framework. . . . 29
2.12 The Cairo architecture diagram. . . . 31
2.13 The MusicMap interface working. . . . 34
2.14 Representation of the apple.com site as a graph. . . . 36
2.15 The TouchGraph interface for the search *web 2.0*. . . . 37
2.16 The Visual Thesaurus interface. . . . 38
2.17 The We Feel Fine applet. . . . 39
2.18 The Web Trend Map 2008 example. . . . 40

3.1 Use cases diagram. . . . 45
3.2 Class diagram for the incidence list. . . . 48
3.3 Screenshot comparison of tetris on System.Drawing (left) and Cairo (right). . . . 54
3.4 The model-view-controller diagram. . . . 59
3.5 Scheme depicting the main classes to be used by the GIC. . . . 60

4.1 The Calliope class diagram. . . . 64
4.2 The Node class. . . . 65
4.3 The Edge class. . . . 66
4.4 The Graph class. . . . 68
4.5 The Athena class diagram. . . . 69
4.6 The Shape class. . . . 70
4.7 The Line class. . . . 71
4.8 The Kratos class diagram. . . . 72
4.9 The Stylesheet class. . . . 74
4.10 The GraphPanel class. . . . 75
4.11 A screenshot of the GIC prototype. . . . 79

4.12 A screenshot of the style sheet editor. 81

A.1 Example of a tree-like schema. 95
A.2 Example of a arrow-like schema. 96

B.1 A class diagram example. 97

C.1 Fruchterman-Reingold layout. 99
C.2 Radial layout. 100
C.3 Random layout. 100
C.4 Orthogonal layout. 101

List of Tables

3.1 Comparative study between incidence lists and matrixes. 47
3.2 Comparative study of layout algorithms. 50
3.3 Brief comparison of graphic libraries. 52
3.4 Time comparison between System.Drawing and Cairo. 54

Abbreviations and Symbols

API	Application Programming Interface
BSD	Berkeley Software Distribution
CDG	Compound Directed Graphs
CRUD	Create, Read, Update and Delete
DAG	Directed Acyclic Graph
DOM	Document Object Model
FR	Fruchterman-Reingold
GDI	Graphic's Device Interface
GIC	Graph Interface Control
GISA	Gestão Integrada de Sistemas de Arquivo
GPL	General Public License
HTML	Hyper-Text Markup Language
IDE	Integrated Development Environment
ISO	Internation Standards Organization
JPG	Joint Photographic Group
JUNG	Java Universal Network/Graph Framework
LGPL	Lesser General Public Licence
MVC	Model-View-Controller
NP	Norma Portuguesa
OOUI	Object Oriented User Interface
OS	Operative System
PDF	Portable Document Format
PERT	Program Evaluation and Review Technique
PNG	Portable Networks Graphics
PS	Post Script
URL	Uniform Resource Locator
XML	Extensive Markup Language

Chapter 1

Introduction

The introductory chapter makes a brief description of the paper's project — a graph interaction control. Work context, project description, motivation, objectives and paper structure are covered.

1.1 Project Context

This paper presents the investigation and development of a graph interaction control during the Project course of the fifth year of the Mestrado Integrado em Engenharia Informática e Computação from Faculdade de Engenharia da Universidade do Porto. The project had a duration of twenty weeks during the second semester of the scholar year of 2007/2008 and was conveyed through an internship in a software company – ParadigmaXis S.A.[1].

ParadigmaXis S.A. (henceforth pX) is a software and architecture engineering company based in Porto. It was founded in 2000 and works mainly on the areas of archive management, information system's safety and geographically referenced information visualization. The graph interaction control described in this paper is planned to be part of one of pX's largest products - GISA [2]. GISA stands for *Gestão Integrada de Sistemas de Arquivo*, and it is a commercial software application oriented to the management of large archive systems. The application is based on international standards, and is built on top of an integrated model that focuses on the representation of several forms of information, including thesaurus and structural organograms.

The graph interaction control itself deals with the following areas of computer science: software architecture, graphic computation, information visualization and human-machine interaction.

1.2 Project Description

The project consists in the creation of a generic graph control for desktop applications that allows visualization and interaction with information in the form of graphs. The visualization of this information should be rich and customizable, so that different types of graphs, nodes and edges can represent different entities accordingly. The control should provide means to create and edit new graphs and to navigate existing ones, loaded from a database. Usability is the main quality factor: the control should strive to provide a user-friendly interface.

The final version of the control should also be platform independent and work at least in Windows and Linux environments.

As stated previsouly, the graph interaction control encompasses a wide range of fields — graph theory, information visualization, human-computer interaction, and software engineering are the most important ones.

1.3 Motivation and Objectives

The main motivation for this project was to develop a control that could enhance and facilitate the understanding and reading of two specific concepts used in GISA: thesaurus and organograms (refer to subsection 2.3).

Thesauri are normally represented (both digitally and in printed form) as textual lists, and understanding the underlying structure that connects different terms is often difficult. Organograms, on the other hand, provide an interesting challenge in the form of developing an automated way to draw them in an aesthetically pleasant way. Graphs not only provide dynamic structures with the potential to represent any number of entities and the way they interact with each other, but they have also been the subject of developed work and studies that allow for automated layouts, in-depth searches and other tools. Thus the motivation for the graph interaction control was to find a way to use existing techniques and visualization processes to represent thesaurus and organograms in an automated and aesthetically pleasant way.

There are many good applications out in the market that are graph oriented, but they normally focus on specific niches. The need for truly reusable tools for graph editing and interaction was thus also a big motivation.

The objectives set by pX for the control include:

- Graph loading, creation and editing;
- Automatic layout of graphs to aesthetically pleasant forms;
- Broad range of visual properties to allow customizable graphs;

- Standards-compliance with any related standards;
- User-friendly interface and easy-to-use tools;
- Use of an open-source graphic library;
- Multi-platforming to at least Windows and Linux;
- Modular architecture to allow reusability and expansibility;
- Written in C# code, to ensure compatibility with GISA.

However, this is just a brief summary. An extensive list of objectives can be found in section 3.1.

1.4 Paper's Structure

Aside from this introductory chapter, this project report is divided into four major chapters, each depicting a different part of the development of the graph interaction control. Chapter 2 presents a literature review of the state of the art on several fields related to graphs and graph-oriented applications. Chapter 3 focuses on listing precisely the objectives, explaining the decisions and defining the methodology and the architecture for the project - it serves as a high-level, theoretical description of the project. Chapter 4 summarizes the implementation of the prototype: it presents the libraries, classes and the prototype developed for the project; it is a technical summary of how the problem was solved. Finally, chapter 5 analyzes what objectives were and weren't fulfilled, sets a road map for future development and draws some final conclusions about the project.

Chapter 2

State of the Art

This chapter makes an overview of the state of the art for both graph drawing theory and graph-oriented applications. It is important to understand that developing a graph interaction control is something that encompasses a broad spectrum of technological areas: graph drawing techniques, graphic libraries, graph layout algorithms, interaction paradigms and graph data models are all important areas, each becoming subject to decisions based on what is better to the project's future and objectives.

With this in mind, the chapter is divided into several subsections, each building on top of the previous and adding some insight into another area of work. Section 2.1 makes an introduction to graph theory; section 2.2 discusses graph drawing; section 2.3 presents the concepts of thesaurus and organograms, and how they will fit into this project; section 2.4 introduces two possible data models to represent graphs; section 2.5 presents several graph layout algorithms and takes some conclusions about each of them; section 2.6 reviews existing graphic libraries that could be used in the project; finally section 2.7 presents some of the existing graph oriented applications;

2.1 Graph Theory

The first point to be taken into consideration in this project is the understanding of what a graph is. Since most of the project revolves around working with such structures, it is important to comprehend the most common graph terms and concepts. This chapter starts by presenting a summary of graph theory and of some simple concepts and formulas.

Graph theory exists since the 18th century, with Leonhard Euler's (1707–1783) *Seven Bridges of Königsberg* [3], which addresses a mathematical problem using a set of bridges on the German city of Königsberg. The problem was to find a path that would cross the seven bridges once and only once on each bridge, as shown in Figure 2.1. Euler formulated the problem replacing each landmass as a dot, or node, and replacing each bridge as an edge connecting two nodes. He then used the concept of a node's degree to prove that such path is in fact non-existent.

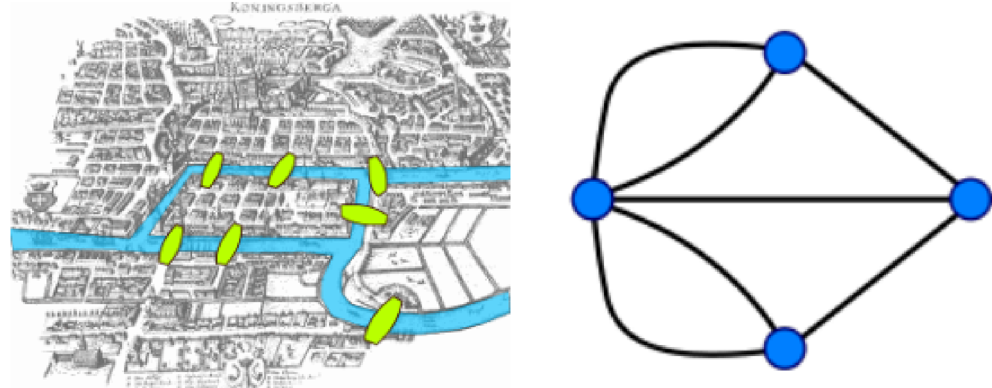


Figure 2.1: The Bridges of Königsberg problem.

Since then, the interest in these conceptual structures has been increasing. Nowadays graphs can be used to represent information as diverse as molecular structures or information flowcharts. The appearance of areas such as graph drawing and the use of graph as a mean to represent sets of interconnected information both in a visual and mathematical way have contributed to the large amount of work developed in this area. Visually, a graph can be presented in many different forms, such as the one in Figure 2.2.

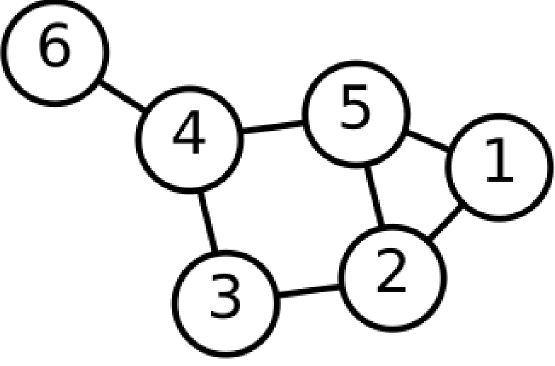


Figure 2.2: A simple example of a graph.

A very precise and complex introduction to graph theory is given in [4], which provides an excellent reading on the subject. However, and since most of the concepts introduced in said book are beyond the scope of this work, this chapter follows closely the introduction texts from [5] and [6], as they present significantly fewer concepts.

Mathematically speaking, a *graph* $G = (V, E)$ is a pair of sets such that V is a set of *nodes* or vertices, and E is a set of 2-element subsets of V called *edges*. In a graph, edges connect different nodes, creating a network of information or relation between concepts. If loops are allowed (a node connects to itself), then the maximum cardinality E is defined as $|E| = |V|^2$. A non-looping graph can have at most $V(V-1)$ edges. The number of nodes in a graph G is called the graph's *order* and is represented as $|G|$. The number of edges in a graph G represents its *size*. A node v is considered to be *incident* to an edge e, if $v \in e$. Two edges $e \neq f$ are incident if they have at least one node in common.

The degree of v is $d(v)$, and is defined as the number of edges that start or end a $v(|E(v)|)$. If a node v has $d(v) = 0$, it is considered *isolated*. On the other hand, if all pairs i, j of nodes are connected by an edge the graph is considered *complete*. A *path* can also be defined as a set $V' = \{i_0, i_1, ..., i_n\}$ of nodes connected by the set of edges $E' = \{\{i_0, i_1\}, \{i_1, i_2\}, ..., \{i_n - 1, i_n\}\}$. A path connects two nodes through a set of edges that crosses each node once only. If $i_0 = i_n$ we have a closed path, called a *cycle*. The *complement graph* of G is $G^C = (V, E^C)$, where $E^C = V^2 \backslash E = \{\{i, j\} | \{i, j\} \ni E\}$. That means the graph's complement is obtained by connecting all edges that aren't adjacent with an edge, and removing all current edges from a graph G. Finally, graphs can have a *weight* (or cost) associated with each edge. A *weighted graph* is defined as $G = (V, E, \omega)$ where $\omega \colon E \to \mathbb{R}$.

This introduction assumes that the graph has undirected edges, in the sense that an edge $\{i, j\} = \{j, i\}$. A *directed graph* (or *digraph* from here on) can be defined as a graph where an edge has a specific order, such that $\{i, j\} \neq \{j, i\}$. Normally, these edges are represented with a small arrow at the end tip of the edge. Directed edges represent a path that can be only taken in a given direction, and not the other way around. A node v in a digraph has an *indegree* $d_{in}(v)$ that is the number of incoming edges to v, and an *outdegree* $d_{out}(v)$ that is the number of edges starting in v; we can also say that $d(v) = d_{in}(v) + d_{out}(v)$.

Most graphs vary from the above in small ways: they are called *mixed graphs* when they contain both directed and undirected edges, or *simple graphs* if they contain only one type of edge; a node may have an edge connecting it to itself, in which case it is called *looping graph*; also, if a graph has no isolated nodes ($\forall v \in V, d(v) \neq 0$), all the edges are directed and there are no cycles whatsoever, it is called a *tree*. Trees are special graphs, and are often treated differently in terms of

visualization.

Finally, it is important for this work to define the *density* of a graph for purposes of finding out the most suitable data structure to define a graph. While the immediate ratio between the cardinality of V and E can be defined as $\varepsilon(G) = |E|/|V|$, this will only give a measure of how many edges there are in average for each node. The density of a graph represents the proportion between the number of edges and nodes, and is defined for an undirected graph as $\frac{2E}{V(V-1)}, 0 \leq D \leq 1$. If $D = 0$ there are no edges, and if $D = 1$ the graph is complete. A graph can be considered either *sparse* or *dense*. Figure 2.3 shows a sparse graph and a relatively denser one. While there are no specific rules as to when a graph is to be considered either [7], it is generally accepted that in terms of computational costs, a graph G is sparse if $|E| = \Theta(|V|)$ and dense if $|E| = \Theta(|V|^2)$. This means that if a graph is near to be complete, and the number of edges is close to $|V|^2$ we have a dense graph, otherwise the graph is sparse.

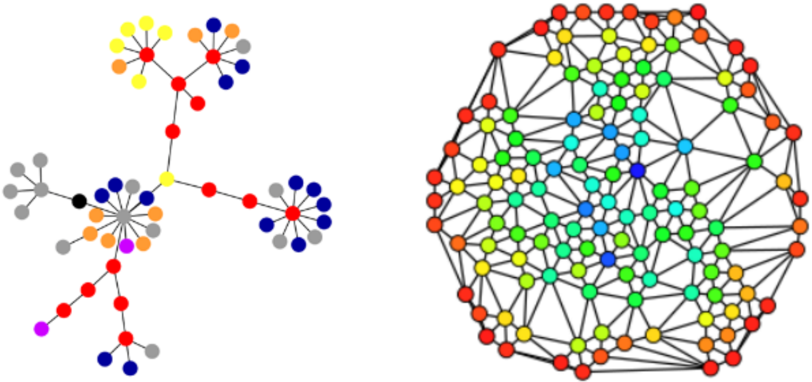


Figure 2.3: A sparse graph (left) and a dense graph (right).

2.2 Graph Drawing

Equally important to understanding the mathematical definition of a graph is the ability to draw them in a readable way. This is not always easy, specially when the graph at hand has a large number of nodes, is too dense or contains additional information elements such as a text identifier for each node. Hence, when graphs are drawn for any model, conceptual and design conventions are defined to ensure both consistency and readability between them: conceptual conventions impose certain restrictions on the way a graph is drawn, while design conventions are used to make it look aesthetically pleasant.

Graph drawing is a broad concept, once that encompasses areas as diverse as graph theory, geometry, topology, visual languages, perception, information visualization or graphic design. There is an annual Graph Drawing symposium [8] with extensive literature about all these areas individually and how they all relate to graphs. In section 2.5, an important part of graph drawing (layouts) is further discussed. In this section, however, the discussion will be about graph drawing in terms of visual languages and information visualization. In other words, the discussion is about ways to draw graphs. The article [9] provides a good reading to understanding graph drawing and constraints. [10] also provides an interesting insight into how to draw graphs, although from a heuristic point of view. For the specific case of visual representations of thesauri and organograms, the interested reader can look into [11] and [12].

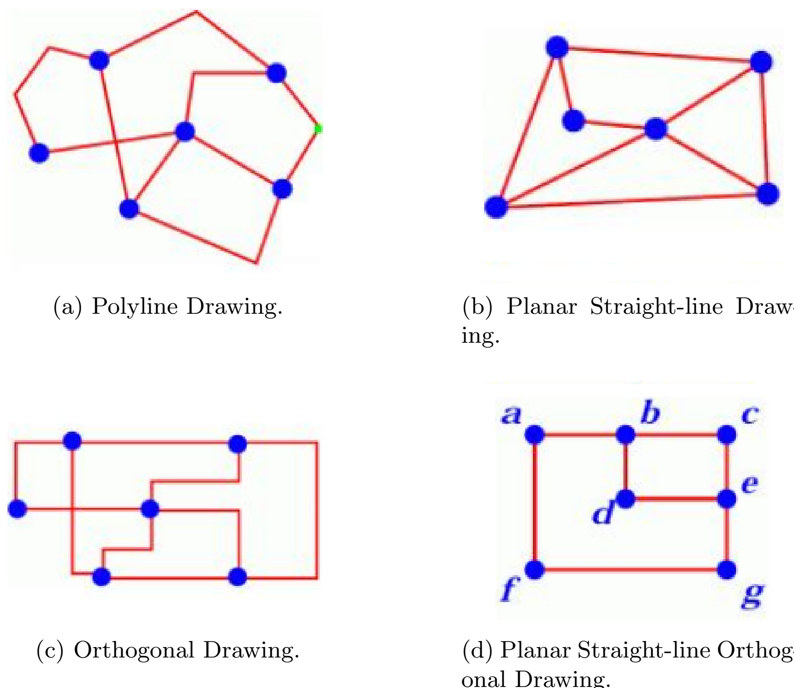
Graph drawing can be divided in two major aspects: drawing constraints and drawing conventions.

Constraints are limitations imposed on the way the nodes and the edges can be drawn; they limit the geometric representation of edges and nodes in a graph. One can say, for example, that all edges must be drawn as straight-lines, never using bends or curves. One can limit the placement of nodes to a standard grid. Or one can establish that directed edges should point in a common direction as much as possible. The most common constraints are:

- Polyline (fig. 2.4a): edges can have bends or curves;
- Straight-line (fig. 2.4b): edges must be a straight-line;
- Orthogonal (fig. 2.4c): edges must be drawn as orthogonal lines;
- Planar (fig. 2.4d): edges can't cross each other;
- Grid-like: nodes must be placed at specific grid-like coordinates.

Figure 2.4 was taken from [9], and depicts the four first constraints in the same graph. Of course, one graph can stipulate more than one of these constraints, save a few exceptions. Not all graphs are planar, and in such cases applying planar constraints will only result in a solution with the least possible amount of crossing.

Drawing conventions specify how the basic and additional elements alike should be drawn for a specific graph. They will determine aspects like the form, color, location of additional elements or whether edges lines are drawn filled or dashed. There is, however, no specific standard on what should be used for any specific case. Hence, different drawing conventions applied to the same graph will result in a different appearance. The following is a list of possible parameters that may vary in a graph's appearance:



(a) Polyline Drawing.

(b) Planar Straight-line Drawing.

(c) Orthogonal Drawing.

(d) Planar Straight-line Orthogonal Drawing.

Figure 2.4: Types of graph drawing.

- Node: pertaining the aspect of individual nodes.
  - Form: circle, square, ellipse, etc;
  - Outline: presence or not of an outline;
  - Fill Color: the color to fill the node;
  - Outline Color: the color of the outline;
  - Size: relative size of the node;
  - Text: presence or not of a caption.
    - Size: size of the text;
    - Position: position of the text relative to the node;
    - Alignment: text alignment;
  - Others...
- Edges: pertaining the aspect of edges.
  - Line-type: the type of line, such as single, dashed, double, etc;
  - Color: the color of the line;
  - Thickness: the thickness of the line;
  - Value: the presence or not of a weighted value for the edge;

  - Text: presence or not of a caption.
    * Size: size of the text;
    * Position: position of the text relative to the edge;
    * Alignment: text alignment;
  - Others...

- Graph: pertaining the aspect of the graph as a whole.
  - Size: the size of the graph;
  - Border: the presence or not of a border;
  - Back color: the back color of the graph;
  - Border color: the color for the borders;
  - Others...

This list is in no way complete. Conventions can be set for different types of edges, or for different degrees of distance to a central node. In Figure 2.5, two different graph conventions result in different appearances for the same graph. On the left, all edges and nodes are black. On the right, second-generation edges are dashed, the edge color is red, and the node color is orange, with a shading effect.

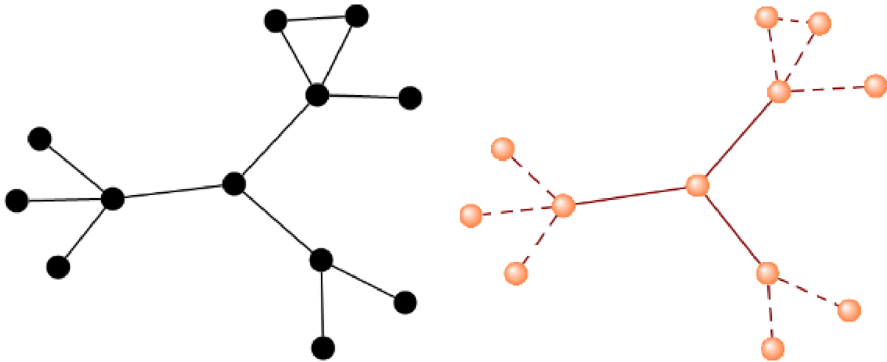


Figure 2.5: Different graph conventions applied to the same graph.

There are several quality metrics that may apply to the drawing of a graph. One of them is the visual aspect of a graph. Special care should be put into using the best set of constraints and conventions to create a readable graph.

2.3 Thesaurus and Organograms

Since the project described in this article will deal with the two specific cases of thesaurus and organograms, the following subsection will present the most common drawing conventions for said structures, along with some examples.

A thesaurus is an indexed compilation of *concepts* and relations amongst themselves. They are normally ordered in an hierarchical way, such that broad-scope terms (known as the *domain of knowledge*) act as roots. Thesaurus can also be a compilation of *expressions*, but this is beyond the scope of our work. Each term has a list of relationships to other related terms. These relationships can be of three general types: *hierarchical*, *equivalence*, or *association*.

Hierarchical relationships represent connections between broader or narrower terms. One may say, for example, that *vehicle* is a broader term than *car*. Reciprocally, *car* is a narrower term than *vehicle*. These types of relationships are always bidirectional. Equivalence can be used to refer a similar or equivalent term with the same meaning. One may say that *freedom* is equivalent to *liberty*, or that *CEE* and *EU* share a temporal relationship (*CEE precedes EU in terms of time*). Finally, relations of association are often used to connect two terms that are related neither hierarchically nor equivalently. One may say, for example, that *Company A* is associated with *Company B*, and that often means that if the user is looking for one term, it might be useful to look into the related term too.

Thesauri are normally organized into indexed alphabetical lists, simplifying the search by presenting information in a structured simple way. Representing thesauri graphically, however, poses a much harder challenge. However, there is an international standard (ISO 2778, and the loosely based Portuguese version NP 4026 [12]) for the establishment and development of monolingual thesauri that provides some guidelines on what should be taken into consideration when drawing a graphical representation. It distinguishes two types of diagrams:

- Tree-like schema;
- Arrow-like schema.

Both types present only a small descriptor or identifier for the term. There is no need to give each term a specific symbol, although the position may be determined in a coordinate-like system. Although this standard is focused mainly on thesaurus representation on actual paper, this may nevertheless be a good inspiration for human-machine interaction with a virtual thesaurus.

The tree-like schema uses orthogonal undirected and straight-line edges to relate terms in different levels. It resembles a tree, with a root term that is part of the domain of knowledge, and all terms in it must be related hierarchically. Narrower terms are placed beneath broader terms. The graph should be planar whenever possible, and avoid multi-level edges as much as possible. No considerations are made regarding equivalence or association relationships. This can be seen as an tree-like representation of a thesaurus.

The arrow-like schema, places the broader term in the center of the schema, with an emphasized identifier. Subsequently narrower terms are placed around the center. The narrower a term is, the farther it will be placed from the center, creating layers around the broader term. Hierarchical relations are drawn with an arrow, and association relations are drawn with a dashed line, pointing out of the diagram. No considerations are made regarding equivalence relationships. This could be seen as a graph representation of a thesaurus.

Appendix A presents an example of each of these types of diagrams. The diagrams are loosely based on the ones presented in the NP 4026.

Organograms are a special case of graphs. They have the same basic structure of a graph, yet they are never closed. They are normally directed, with a root term on top and subsequently subordinate terms in lower levels. They may be used to represent the internal structure of a company, for example. They are normally represented using orthogonal lines and rectangles for concepts or entities. They also have a fourth type of relationship, used to specify a chronological order between terms. It is called a *temporal* relationship.

2.4 Data Structures

In this section, two ways of implementing a data structure for the purpose of keeping information about a graph are presented. Since a graph is basically a set of nodes and edges between them, we will define two basic objects to represent graphs: Nodes and Edges. An object node $n$represents information about a node, such as its caption, unique identifier or its list of edges. An edge e represents an edge and its related information, such as weight, caption, and starting and ending node. Additionally, one may define a third container object Graph, that contains instances of the nodes and edges on a given graph. An example of such object implementation can be found in [7]. The course notes [13] and [14] also provide some insight into this subject.

According to [15], there are two ways of representing a graph in a data structure: an adjacency list or an adjacency matrix. The two main differences between them are the data types used to represent the information. An adjacency list uses a collection of arrays, while the adjacency matrix uses a matrix. Furthermore, each type has its advantages and disadvantages, depending on the density of the graph and on the type of operations one may want to apply to the representation.

2.4.1 Adjacency List

An adjacency list (sometimes referred to as an *incidence list*) is simply an array A composed of a set of $|V|$ lists, one for each node in the graph, as shown in figure

2.6. The nodes are normally stored in an arbitrary order. The list $A[u]$ is filled with all the adjacent nodes to u, or alternatively, with pointers to said nodes. For undirected graphs, this means the space required for the lists of adjacent nodes is $2|E|$ since a connection (i, j) will be present in both node's list. If the graph is directed, however, this space is only $|E|$. The total amount of memory required for this list is thus $\Theta(|V| + |E|)$. Should one want to keep information not only about the nodes, but also about the edges, one can fill each list from A with pointers to edge objects instead.

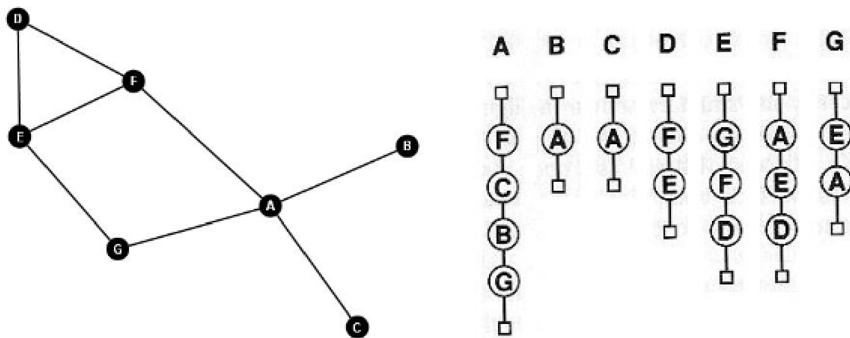


Figure 2.6: A graph and it's corresponding adjacency list.

The main advantages of the adjacency list are its low memory cost and also its low computational cost for some operations. Finding all the adjacent nodes to n, for example, takes $\Theta(1)$ time. Adjacency lists are also better used when a graph is considered sparse, since they will spend strictly the memory needed to represent each node once.

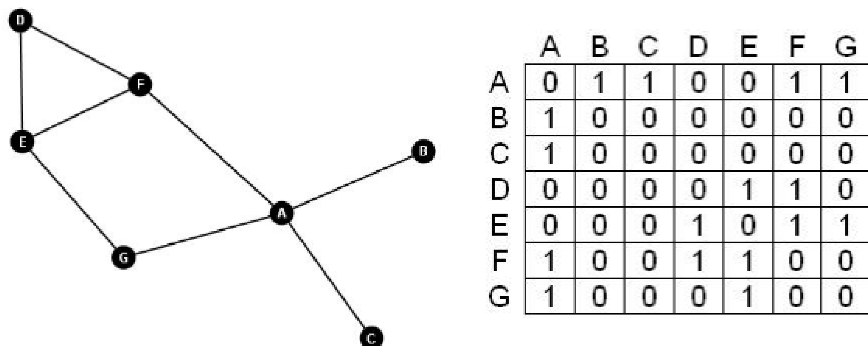
2.4.2 Adjacency Matrix

The idea of the adjacency matrix is to use a grid-like structure to determine where the edges are, as shown in Figure 2.7. An adjacency matrix is a matrix A of dimension $|V| \times |V|$ where cell $A_{ij} = [i][j]$ is such that:

$$A_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

Hence we have a matrix where edge (i, j) is represented by a 1 in cell $A[i][j]$. The adjacency matrix requires $\Theta(|V|^2)$ memory space for any given graph. This is assuming the edge information will be represented by a bit. If we want to keep information about each of the edges, each cell will lodge an edge object, costing significantly more. If the graph is undirected, the adjacency matrices for graph G and its transpose G^T will be the same, since an edge is represented both at $A[i][j]$

and at $A[j][i]$. In this case, we can halve the memory needed by keeping only the information of either of the triangular matrices of the adjacency matrix.



	A	B	C	D	E	F	G
A	0	1	1	0	0	1	1
B	1	0	0	0	0	0	0
C	1	0	0	0	0	0	0
D	0	0	0	0	1	1	0
E	0	0	0	1	0	1	1
F	1	0	0	1	1	0	0
G	1	0	0	0	1	0	0

Figure 2.7: A graph and it's corresponding adjacency matrix.

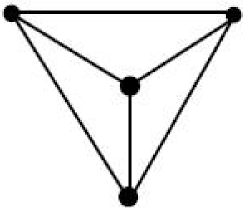
The advantages of this structure is mainly its use for dense graphs. If the graph is almost complete (i.e, if $|E| \approx |V|^2$), one can consider it dense. In this situation, the adjacency matrix will still only need $\Theta(|V|^2)$ memory, while the adjacency list will now need $\Theta(|V| + |V|^2)$. In this case, the adjacency matrix is much more efficient than the list. Although there is no exact boundary to determine when a graph is dense, it has been calculated that above a density of $\frac{1}{64}$, the matrix representation becomes less memory consuming. This analysis, however, assumes that only connectivity is being stored, and for situations where we also intend to keep information about each of the edges, this value is bound to be higher.

Adjacency matrices also behave differently than lists in some types of operations. While finding out all the adjacent nodes to n only takes $\Theta(1)$ time in a list, it takes $\Theta(V)$ on a matrix, since a whole row must be checked. On the other hand, checking if there is an edge between two nodes i and j takes $\Theta(min\{d(i), d(j)\})$ in a list representation, but only $\Theta(1)$ in a matrix.

2.5 Graph Layout Algorithms

A plethora of layout algorithms for graphs have been developed over the last twenty years. They apply to different types of structures, from regular binary trees to digraphs, and involve drawing graphs so that they are easy to read and understand [16]. This is normally achieved by defining a set of aesthetic metrics that act as optimization goals for any algorithm. For generic graphs, the most common aesthetics to look for are:

- Display of symmetry (if existent);



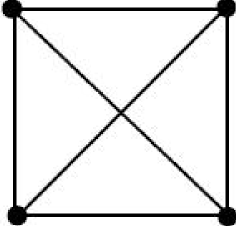


Figure 2.8: Different aesthetic criteria result in different graphs.

- Planarity (avoiding edge crossings);
- Linearity (avoiding bends in edges);
- Node uniformity (balanced distribution of nodes);
- Edge length uniformity (balanced medium length of edges).
- Direction (for directed graphs, edges should point in the same direction).

While trying to achieve as many of those criteria as possible, it is generally impossible to conjugate them all. In Figure 2.8 (as found in [17]), the left graph is planar and contains some degree of symmetry. However, the same graph can be drawn as in the right, with symmetries being maximized at the cost of planarity. Seldom can we get the best of both worlds. It is also important to determine the computational cost of each of the criteria, and the final purpose of the graph. If a graph is merely to be rendered and used as a static image, then planarization could be one of the priorities and extra computational cost could be dispensed. Otherwise, if a graph is to be used interactively, time is a major constraint, and some high-cost aesthetic criteria must be sacrificed.

Graph-drawing algorithms can be divided into three major groups: algorithms for straight-line drawing, for planarization, and for polyline drawing. The first group provides general algorithms that focus on aesthetics such as symmetry, edge distribution and fast computation, using only straight lines. The second group are high-computation cost algorithms that try to draw a graph as planar as possible. The last group focus on the use of dummy nodes to achieve a pleasant result, with the dummy nodes being removed and replaced by polylines in the end.

For the purpose of this report, and since the project will deal almost exclusively with straight-line graph drawing, only a simple explanation of how the major algorithms in this area work will be presented. A slightly more complex algorithm for polyline graph drawing that might be useful for drawing organograms will also be described.

The article [16] provides an extensive list of graph-drawing algorithms in all its forms. In [17] there is also a brief description of the most important algorithms for straight-line graph drawing.

2.5.1 Force Directed Spring Model

Considered by many as the first straight-line graph layout algorithm, the spring model presented by Eades in [18] dates back to 1984, and was developed as an *embedder* for one of Eades' previous works. The objective was to create a simple heuristic to assign coordinates to each node in a graph, in a way that the final result would be somehow aesthetically pleasing. The author notes that "aesthetically pleasing" is a subjective concept, and that as so, this task is a formidable one. The criteria set by Eades are similar edge length and symmetry display, if it exists.

Perhaps the best way to explain the concept of the algorithm is to present the author's own metaphor:

> "The basic idea is as follows. To embed a graph we replace the vertices by steel rings and replace each edge with a spring to form a mechanical system. The vertices are placed in some initial layout and let go so that the spring forces on the rings move the system to a minimal energy state. The algorithm outputs the positions of the vertices in this stable state."

To model this system, logarithmic strength springs are used, instead of linear strength springs like stated in Hooke's Law. This deviation from reality is due to the fact that linear strength makes forces too strong when vertices are too far apart. Hence, the force created on a node by a spring is dependent of the length of the spring d and of two constants $C1$ and $C2$ as follows:

$$F_a = C1 \times \log(\frac{d}{C2}) \tag{2.2}$$

Since $d = C2 \rightarrow F_a = 0$, $C2$ represents the desired length for a spring. This force is applied to all the adjacent edges. A second repellent force is also calculated between all non-adjacent edges. This force is dependent of the linear distance d between said edges and of a constant $C3$, and is given as:

$$F_r = \frac{C3}{\sqrt{d}} \tag{2.3}$$

A fourth constant $C4$ is also defined to adjust the displacement of each particle. He suggests that by experiment, $C1 = 1, C2 = 1, C3 = 1, C4 = 0.1$ works well with most graphs, although no mathematical proof of this is given. The stopping condition to the algorithm is also theorized to be around 100 iterations into the

algorithm, at which point most systems attain a minimal energy state, although no proof of it is provided. The pseudo-code for this algorithm is as follows:

Algorithm 1 Eades' Spring Layout Algorithm

```
Place vertices of G in random locations;
repeat
  for all v ∈ G do
    Calculate F_a for v;
    Calculate F_r for v;
    Calculate F_V from F_a and F_r;
    Move vertex C4 × F_v;
  end for
until N iterations have passed
```

The algorithm is pretty simple in its own way, and has the disadvantage of resulting in different layouts for different initial placements. The existence of a small heuristic of initial placement to help the spring algorithm achieve a pleasant result could be used as an optimization. In terms of costs, and since each iteration treats nodes independently, calculating the force on one node costs $\Theta(|V|^2)$, and this is cycled through all of the nodes, costing $\Theta(|V|^3)$ for each iteration of the algorithm. Still, Eades argues that for small sparse graphs with less than 30 nodes, the running time is almost unnoticeable. It is also stated that the algorithm runs quite faster on graphs with no cycles (trees), and that the number of iterations in these cases could be substantially lower. On the other hand, experience has shown that the algorithm doesn't do so well with dense graphs and graphs with a small number of bridges.

2.5.2 Kamada-Kawai

The Kamada-Kawai algorithm [19] is a direct descendant from the Eades' algorithm described in section 2.5.1. Conceptually, the idea is the same: to replace edges in a graph with springs and to find a state of minimum energy in the system by solving a set of differential equations. The difference here, however, is that Kamada connects all the existing nodes, and not only the ones connecting to each other. Connecting nodes have a smaller spring, but non-adjacent ones also have a spring keeping them apart. This introduced the concept of ideal length between non-adjacent nodes, keeping adjacent vertices closer to each other, but at the same time spreading the graph so it doesn't become cluttered. This ideal distance is proportional to the shortest path between two non-adjacent nodes.

The article [19] provides a good presentation to the algorithm, however both [20] and [9] also make an excellent resume on the mathematical principles behind the layout technique.

Kamada and Kawai stated that the total energy of a graph can be defined as follows

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} \cdot k_{ij}(|(p_i - p_j)| - l_{ij})^2 \qquad (2.4)$$

where $p1, p2, ..., pn$ represent the nodes of a graph, and where l_{ij} is the natural length of a spring between p_i and p_j, defined as $l_{ij} = L \times d_{ij}$. L is the desired length for the edge, and d_{ij} is the length of the shortest path between p_i and p_j. L is normally chosen in function of the size of the area available to drawn the graph.

The energy of the system is then reduced by solving a partial differential equation for each node, in a process that tries to find a local minimum that minimizes the energy of the system. As each node is repositioned in each iteration, the total energy is recalculated and the process is repeated until the system reached a preset threshold. Since only one node is repositioned at a time, recalculating the energy of the system is faster than in Eade's algorithm, as only the contribution of one node needs to be taken into account.

The simplest heuristic to find all shortest paths takes $\Theta(N^3)$ time to compute. Still, if better heuristics are used, and if some bookkeeping is used, the algorithm takes only $\Theta(T \times N)$ where T is the number of inner loops and N is the number of nodes in a graph. This makes it more effective than Eade's algorithm, at least for a vast majority of graphs.

The Kamada-Kawai algorithm reduces the number of edge crossings and displays planarity most of the times. This happens because edge crossings raise the energy of the whole system. The algorithm also tends to reveal symmetries when present, and can also be extended into layered graphs by assigning nodes in the same level a fixed value for the y coordinate and allowing only x to vary.

2.5.3 Fruchterman-Reingold

The Fruchterman-Reingold algorithm (henceforth FR), described in detail in [20], is a modification of Eades' spring-model algorithm. It was designed for simple, undirected graphs with linear connections, and its objective was to produce aesthetically pleasing graphs in as little time as possible, using a simplified physics system, focusing on speed and simplicity. The article describes five essential aesthetic criteria: even distribution of nodes, minimal edge crossing, edge length uniformity, inherent symmetry and containment in the desired frame. While not trying specifically to attain any of these criteria, the FR algorithm does naturally distribute the nodes in an even way, displaying similar edge length and reflecting symmetry when existent.

Instead of simulating a set of springs and the system's energy state, the FR uses a different metaphor, based on the behavior of a set of particles or bodies and the forces that attract and repulse them. It represents an approximation to either a molecular system or a planetary one. The simulation of such a model is normally called a many-body problem [21], and is normally solved by finding the adequate values for the forces within the system. The FR authors, however, derive from the Eades' algorithm that the simulation of the physical system doesn't need to be exact, and they simplify their system in a few ways. Hence, all edges repel each other, but two nodes share an attractive force if they have an edge connecting it.

The other simplifications comes from the fact that the forces in a many-body problem are generally calculated to find a dynamic equilibrium, and in this case a static equilibrium is enough. Instead of finding the *acceleration* derived from actual forces, FR uses the *velocity* given to a particle by the forces acting on it. FR uses a constant k that is defined as the optimal distance between nodes, and is given as

$$k = C\sqrt{\frac{\text{area}}{\text{number of nodes}}} \tag{2.5}$$

where C can be considered 1 most of the times. The value of k represents the radius around each node which can be considered as the free area. If another particle comes into this area, it should be strongly repelled. For a distance of d between two particles, it can be stated that the attractive force f_a and the repulsive force f_r can be defined as

$$f_a(d) = \frac{d^2}{k} \tag{2.6}$$

$$f_r(d) = \frac{-k^2}{d} \tag{2.7}$$

Finally, a global temperature for the system is set, cooling down with time, and limiting the displacement of the forces on each iteration of the algorithm. The cooling function can be defined as a simple inverse linear function from a set start value. The authors suggest a tenth of the frame's width, but different options can be used for best fine-tuning. The algorithm is then divided in three steps: calculating the attractive forces between the particles, calculating the repulsive forces, and finally limiting the particles' individual displacement via the system's temperature.

The basic FR has a computational cost of $\Theta(|E| + |V|^2)$, since an iteration of the algorithm calculates the attractive forces between each adjacent nodes (once for each edge) and the repulsive forces between all nodes. The authors have devised an improvement over the basic algorithm to reduce this cost to $\Theta(|E| + |V|)$: it consists of using a gridded version of the draw area, in which each node is only repelled by

other nodes in a circular area of 2k radius around it. This discards the weaker force effects of farther nodes, and is essentially the same as computing the repulsive force as

$$f_r = \frac{k^2}{d} \cdot u(2k - d) \tag{2.8}$$

where

$$u(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

The FR has also considered how the modeling of the frame's limits should work, as shown in Figure 2.9 (figures 5 and 6 from [20]). When a node has been displaced such that its new location is outside the boundaries of the frame, two solutions can be devised: the component can either be stopped when it reaches a boundary, much like if it was stuck on the limit, or bounce back, which could lead to extensive computation. The authors propose a solution where the component will be stuck when it reaches a boundary, but the displacement that is normal to it will be allowed to continue. Hence, if a component hits the top wall coming from the right, it will stop moving up, but it will be allowed to slide left until that displacement is complete.

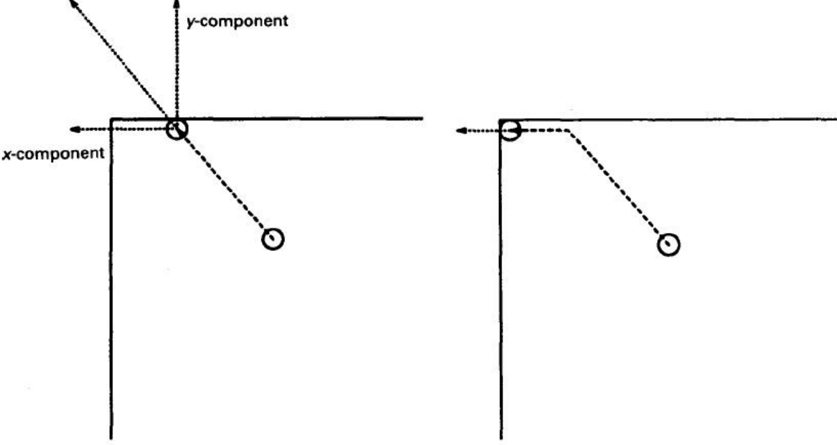


Figure 2.9: Frame limiting in FR algorithm.

The FR is a lightweight algorithm that focus on simplicity and speed. However, this does not come without some cost. Firstly, it is important to understand that the initial configuration of the nodes of the graph will result in unique graphs. Since most programs simply randomize the initial position of the nodes, the results with FR won't always be the same, or even acceptable. However, the output result for a specific graph can then be used as input by the same algorithm to attain

more precise results. Another weakness is the lack of theory to support how many iterations should be executed to get a aesthetically pleasing graph. The authors clearly state that

> "We too can offer little justification for the number of iterations, although we experimented with making it a function of $|V|$ or $|E|$."

2.5.4 Sugiyama-Misue

The Sugiyama-Misue algorithm [22] is a four phase technique for automatically drawing simple directed graphs and, in a later version [15], *compound directed graphs* (or *compound digraphs*, henceforth CDG) in a layered way. The authors, Kozo Sugiyama and Kazuo Misue, claimed back in 1991 that most automated algorithms for graph placement normally focus on ordinary directed and undirected graphs. Their focus on a different graph type has turned this algorithm in one of the most famous layout algorithms, mostly due to its early conception. After all, layered graphs have a practical use in many areas, such as PERT[1] charts, organization diagrams, and anything related to hierarchical drawing.

The algorithm focuses on finding a specific layer for each node, and guaranteeing that nodes can only have edges to their adjacent levels. Although the algorithm has a simpler base concept than those explained previously, implementing each of the four phases can be as complex as any of the aforementioned algorithms for graph layout. Hence, this paper presents only a high level explanation of each of the phases of the Sugiyama methodology for *directed graphs*, as presented in [22].

For the sake of completeness, we explain briefly what a CDG is: CDGs allows two different types of binary relations between nodes: edges can be either *inclusive* or *adjacent*, but they are always directed. If the edge is inclusive, the end node includes the start node. If the edge is adjacent, the end node is adjacent to the start node (and not necessarily the other way around).

Mathematically, a CDG is obtained by joining the inclusive graph $G_I = (V, E)$ and the adjacency graph $G_A = (V, F)$, such that V is the collection of nodes in the graph, E is a finite collection of edges where $(u, v) \in E$ represents an inclusive edge where u includes v, and F is a finite collection of edges where $(u, v) \in F$ represents an adjacency edge where u is adjacent to v. The CDG is then defined as the triple $G = (V, E, F)$ that fulfills the two following restrictions:

- **Restriction 1**: G_I is a tree. This means that there must be a unique path between any two nodes in the tree, and that there is only one parent node, with no inclusive edges ending in it (called the root);

[1]PERT stands for Program Evaluation and Review Technique.

- **Restriction 2**: In a CDG $G = (V, E, F)$ that satisfies restriction 1, there are no adjacency edges between nodes that are already covered by an inclusive edge. This means that there are no double edges.

The original Sugiyama algorithm for directed graphs, which is explained in the following text, is described in [1]. The CDG have their own version explained in greater detail in [15]. There are, however, others algorithms for directed graph layout, such as the one given in [10]. An in-depth review and new approach at Sugiyama-Misue's algorithm is given in [23]. A simple, yet complete review of the algorithm can also be found in [17].

The first phase, called *Cycle Removal*, will remove any cyclic relations between nodes and turn the entry graph $G = (V, E)$ into an acyclic graph $G_a = (V, E)$, considered hierarchical. The authors suggest condensing nodes to remove cyclic connections, or alternatively, inverting the direction of a minimum number of edges to guarantee a directed graph, keeping information about the inverted edges so that the original direction can be replaced once the algorithm is complete. In preparation for the following phases, dummy nodes are inserted into the graph, replacing edges that span more than two consecutive layers. Finding the combinatorial solution to the acyclic problem is a NP-Complete problem on itself.

The second phase, called *Layer Assignment*, assigns each node to a horizontal layer. Using algorithms based on the shortest and longest path between two nodes, each node of the graph can be place in a layer, or horizontal level, accordingly to its distance to the root of the graph. In [24], for example, an algorithm which calculates layer assignment and minimizes the total edge lenght is presented. Although no polynomial time bound has been proven for this phase, some linear time heuristics work well for this problem as well.

The third phase, called *Crossing Reduction*, experiments with the nodes on each layer to minimize the number of crossing edges. Cross reduction is often done on a layer-by-layer basis, each step trying to reduce the number of crossings between a level and the following one. For $L_i = 0$ to $L_i = n$ (n being the number of layers), the position of nodes in layer L_i is kept fixed and variations of nodes in layer L_{i+1} are performed to choose the one that reduced edge crossing the most. The process is repeated until the last layer, and then repeated from the bottom to top. When no further crossings can be removed or there are no edge crossings present, the algorithm stops. Several heuristics have been proposed to solve this problem, such as the barycentric, median, or greedy switch heuristics described in [25]. Another important part of this phase is the so-called *bilayer cross counting* problem. This refers to the counting of the number of edge crossings between two levels, and is

considered also a NP-Complete problem. Some possible solutions are presented in [26].

The fourth and final phase is called *Horizontal Coordinate Assignment* and calculates an x coordinate for each of the nodes in the graph. This assignment must take into the consideration aesthetic factors to provide a final nice result. Edge crossing was already achieved as much as possible in previous sections. The fourth phase focuses on keeping the graph compact, and edges as vertical as possible. It also undoes the temporary changes made in the previous phases and renders the final result: inverted edges are reverted back to its original directions, and dummy edges are replaced with a chain edge that *bends* where the dummy edge was.

All in all, it is hard to estimate the computational cost of the Sugiyama algorithm. Most of the problems in each phase are considered NP-Complete, but for all of them there is extensive research and sometimes several solutions. It is nevertheless a hard to implement algorithm.

2.5.5 Simmulated Annealing

Simulated Annealing derives from a metallurgy technique that consists in heating material and then slowly cooling it off to allow the particles to roam randomly with higher energy and have a better probability of finding some better configuration with less flaws over the cooling process. Davidson and Harel proposed a heuristic algorithm in [27] for the drawing of undirected straight-line graphs. [17] gives a very complete overview of the algorithm and some considerations regarding different parameters. [28] is also suggested for an in-depth look at the mathematical formulas behind the algorithm.

The idea is simple: to use a weighted sum of several components of the graph aesthetic appearance to give an idea of the graph's energy. The higher this energy is, the less favorable a specific configuration is, and the lower the energy is the more acceptable it is. Thus one can search for local minimums to attain an attractive looking graph. This system equation can be defined as $E_t = \lambda_1 E_1 + ... + \lambda_i E_i$ where λ_n represents the the weighted importance of the aesthetic criteria E_n.

For each step of the algorithm, the energy for the current system and for a slightly different one (called a neighborhood system or a perturbation) are considered. If the neighborhood system energy is lower than the current one, the new system is accepted, otherwise it is only accepted depending on a temperature function that is proportional to the variation of energy. Thus, the larger the increase in energy, the least probable it is that the new system will be accepted. If the increase is relatively, small, it has a larger chance of being accepted and different configurations might be looked into.

The temperature function should decrease over time in a geometric manner [17], such that $T_{p+1} = \lambda T_p, 0.65 \leq \lambda \leq 0.95$. This guarantees that larger variations may be initially explored, but that eventually the algorithm will look for a local minimum within the state space. It is generally accepted that a value of 0.75 results in a relatively rapid cooling with decent aesthetic results. The algorithm should be run for a set number of iterations n performing at least $30n$ perturbations in each iteration.

Davidson and Harel have defined five aesthetic criterion, as follows:

- Nodes should be spaced evenly around the drawing area, which ideally can be defined as a 1x1 square. Then it's simulated that they repel each other in a manner that $E_1 = (d_{ij}^2)^{-1}$ where d_{ij}^2 is the Euclidean distance between nodes i and j.

- The first criteria will probably scatter the nodes to the edges of the drawing canvas, so to ensure that the graph is centered, one adds a force of repulsion equal to the last one, between the nodes and the boundaries of the drawing area, such that $E_2 = \sum_{i,j=1..n}(\frac{1}{x_i^2} + \frac{1}{y_i^2} + \frac{1}{(1-x_i^2)} + \frac{1}{(1-y_i^2)})$.

- The third criteria is that if adjacent nodes are placed closer to each other, this will help the overall readability of the graph as a whole. For that, one adds that $E_3 = \sum_{(i,j)\in E} d_{ij}^2$.

- There is also the need to discourage edge crossings, as they difficult the reading of the graph. So the fourth criteria is added as $E_4 = \sum_{(i,j)\in E} s_{ij}$ where s_{ij} is 1 if edges i and j cross, 0 otherwise.

- Finally, edges should not be drawn too close to nodes that they do not belong to, as it may confuse the viewer. For that, a final term is added such that $E_5 = \sum_{(ij)\in E, k\in V-(i,j)} \frac{1}{max(d_0, d_{ijk})^2}$ in which d_{ijk}^2 is the distance that goes from node k to the edge i, j. The d_0 is used as a means to avoid division by 0 when the node is over the edge. This last criteria might be overlooked during an initial phase of the algorithm due to its increased computational cost. Most implementations of the algorithm use this one only for fine-tuning of a specific result.

The relative importance of these aesthetic criterion can be altered to improve readability for specific cases. In some graphs, planarity may be considered a priority, and in others correct node distribution can be considered a major factor. In terms of cost efficiency, this algorithm is very demanding. Even when the location of only one node is changed in each perturbation, there is still the need to update up to $|V|$

nodes and for each of them, up to $|E|$ edges. Some papers, such as [20], commented that Simulated Annealing was completely ineffective for interactive graph drawing.

The algorithm is known for being at least as efficient as Eades' spring model, and sometimes even better with large graphs, but unable to render some types of graphs in a pleasing way [4].

2.6 Graphic API

The Graphic Application Programming Interface chosen for a project will be the tools of the trade for a programmer who wants to develop a graph-oriented solution. They will provide the means to develop a information visualization system, and as so, they should provide high-level of customization and an easy way to use a machine's graphic hardware effectively. In our case, if said API has structures and features that are graph-oriented, the better.

We can distinguish two main types of graphic APIs: high and low level. Each provides different levels of abstraction and control over the graphic hardware of a machine. And while high-level APIs may provide a larger range of features that are graph oriented, low level APIs often possess other important features such as faster rendering or more control over what is drawn on the screen. In this section we will review some of the most prolific open source solutions available out there.

2.6.1 High Level APIs

High-level APIs are focused on the specific needs of a specific domain of knowledge. They focus on details such as data representations for the information of the areas of interest, interaction models with said information, or visualization modes for that interaction. They normally try to provide reusable and extensible classes that can be adapted to each project's needs, together with filtering and layout tools to present the information in a aesthetically pleasant way.

Since the level of abstraction is higher, there are some tools developed specifically for graph handling. The following sections present some of the most important libraries found in this area, regarding graphs and interaction features.

2.6.1.1 Piccolo

Piccolo [29] is an open-source toolkit for the development of 2D structured graphic programs in general, and of zoomable user interfaces in particular. It provides a high-level abstraction for user interfaces, since all the low level primitives and event handling is managed by the library. Everything from raster drawing to event handling is taken care of for the user, who can then create his user interface on top

of this. The library is under the BSD Public License, meaning that commercial work may derive from it. Its name was inspired by its size: piccolo stands for *small* in Italian, and since the releases are about one tenth of the project's predecessor Jazz, the name was found adequate. A comparison between Piccolo and his predecessor Jazz and an in-depth look at its architecture can be found in [30].

Piccolo has three similar distribution packages: Piccolo.Java, Piccolo.NET and PocketPiccolo.NET. The first is based on the Java2D API and is written 100% in Java making it useful for multi-platform applications. Piccolo.NET is written in C#, and built on top of the GDI+[2] API, allowing for full compatibility with the .NET framework. PocketPiccolo is a different version for the Compact .NET Framework, making the library viable for PDA-based applications.

The library uses a scene-graph model that keeps a hierarchical structure of elements drawn on the canvas. Each of the elements has an associated transform matrix that allows it to be scaled, translated and shaped around without interfering with others. Additionally, Piccolo also has a camera element, that can be altered to change the way a specific scene is viewed. This allows for a very decoupled object oriented model, where a programmer can simply take the base elements from Piccolo and extend them to their own purposes.

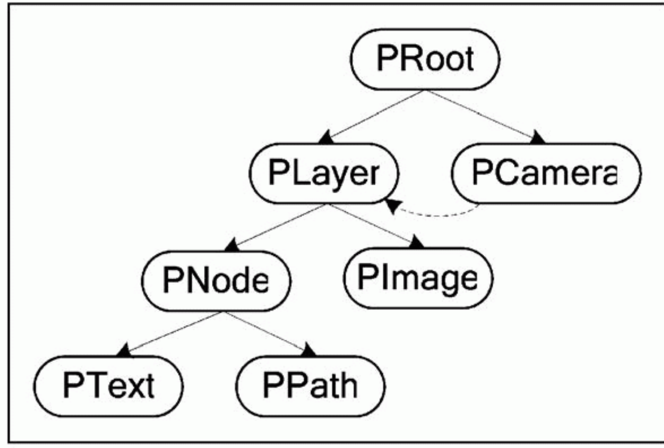


Figure 2.10: The Piccolo object structure.

Piccolo is built upon a few basic element types, as shown in Figure 2.10. A *PRoot* is the basic element that represents the canvas itself. Every *PRoot* has at least one *PCamera* and one *PLayer*. The camera element represents a specific view over the canvas, whereas a layer represents a specific set of graphic elements that can be clustered and treated as part of the same structure. Layers can be ordered along the z-axis, disabled, or effected individually, preserving elements not in it. These

[2] GDI stands for Graphics Device Interface.

elements can be either *PImage* or *PNode*, the former representing an image either read from a file or built with other primitives, and the latter being a generic object that can be extended. These nodes create the scene-graph itself, and can also be distinguished as *PText* or *PPath*: the first may be used to write text output to the screen, and the last to draw a specific path on the screen.

The project authors have reduced their effort on this project as of November of 2006. The library is stable and still used by many projects, but the public support is now limited to a mail group.

2.6.1.2 Netron

The Netron [31] Diagramming Library is a generic graph visualization and layout kit for the .NET platform, under the GPL Public License, and built on top of Microsoft's GDI API. It was originally under the LGPL license, but the author of the kit eventually changed it to GPL and also released a reviewed version that can be purchased for commercial use. The latests releases also come bundled with the *Cobalt IDE*, a customized development environment by the same author. The Netron library was however discontinued, due to the author's focus on his recent *Unfold* [32] library that focuses on the new Microsoft's Presentation Foundation paradigm. There are no papers on this tool, and the information available is strictly that found in the official website.

The library does feature a large set of layout algorithms, such as force-directed layouts, Fruchterman-Reingold, radial tree layouts and others. Graph analysis algorithms include Kruskal, Dijkstra, Floyd and others. Filtering is also accessible through panning, layering, grouping, z-ordering and other functionalities.

2.6.1.3 JUNG

JUNG [33] is a software library developed in Java for the modeling and visualization of data that can be represented as a graph or network. Since it is focused on graphs, it implements a set of defined classes and models that are specifically graph-oriented and thus allow for a plethora of different graph-related algorithms to be easily applied to data. The objective is to allow for a programmable, extensible and easy to use framework that can either be used as basis to other programs or be extended to allow for even more customization. The API also provides a large set of algorithms for everything graph related, from layout techniques to in-depth search of graphs, as well as tools to filter information graphically and in data structures. It is an open source project, under the BSD Public License. An introduction to JUNG can be found in [34].

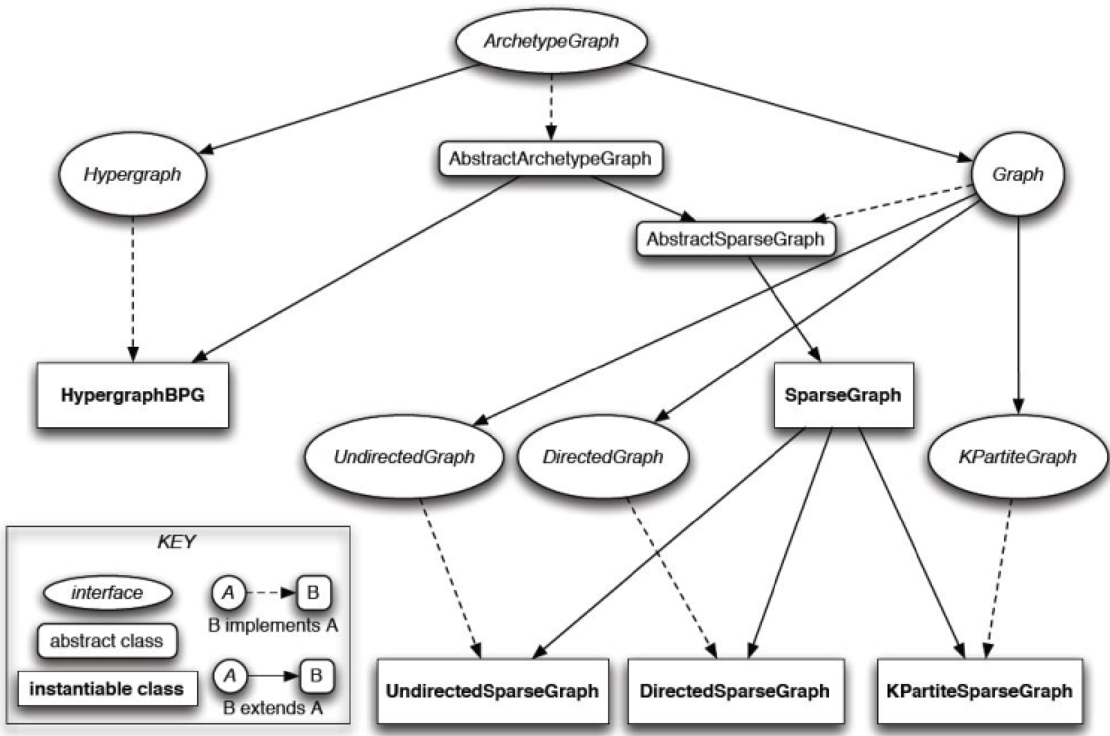


Figure 2.11: Graph types in the JUNG framework.

JUNG's biggest strength is its inner class model. It is completely graph-oriented and include implementations of simple graphs, sparse and dense graphs, hyper graphs or directed and undirected graphs. Nodes and edges also have a myriad of implementation, depending on the graphs they are attached to. A generic diagram for graph classes is shown in Figure 2.11.

In terms of layout algorithms, JUNG provides the following: Fruchterman-Reingold, Kamada-Kawai, Eade's Spring algorithm and the Self-Organizing Maps for large networks. JUNG also provides other algorithms for clustering, neighborhood testing, connectivity problems, maximum flow, centrality and others.

2.6.1.4 Prefuse

Prefuse [35] is a software library built using the Java2D graphic library, and under the BSD Public License. Its intention is of creating a single, reusable and extendable information visualization user interface toolkit. The idea is to have extensive coverage of current visualization models in a large spectrum of areas, while still allowing programmers to creatively develop new visualizations. It is a high-level API that goes beyond interaction with data structures and focuses instead on the presentation, filtering and batch control of large sets of structured data. As so, it provides

a large library that includes layout algorithms, navigation and interaction support, integrated search, object modularization and others, as well as primitive implementations for visual representation of graphs, trees and networks. [36] provides an in-depth look into the architecture of prefuse.

Prefuse uses canonical representations to represent data. The basic data element type is *Entity*, which can be further refined as *Node*, *TreeNode* or *Edge*. These items are then represented visually by *NodeItems*, *EdgeItems* or *AggregateItems* for sets. These types are kept by a *ItemRegistry*, that keeps track of all the state information on a specific visualization. *Actions* can then be applied to specific items in the item registry, via *ActionLists*; these actions include layout algorithms, interaction with pieces and event handling, amongst others . Finally, a *Renderer* can be applied to the item visual representation to model aspects such as color, appearance or position on the screen.

2.6.2 Low Level APIs

Low level APIs are those that cater specifically to the rendering needs of programs. They normally deal with a lower level of abstraction and its objects and concepts are normally directed at graphic representations such as lines, colors, coordinates or fonts. Some use *states* to keep information about important properties of the library, and most of them provide no support for interaction with the drawing contents. In fact, graphics drawn using these APIs are not considered as individual objects, but only as a rendered image on a specific canvas.

Since these APIs are intended to be highly reusable and non-focused, there is significantly less work developed at this level specifically for graphs. We now make a small analysis at some of the main low level graphical APIs both related and non-related to graphs.

2.6.2.1 System.Drawing

The System.Drawing namespace [37] is a library that provides access to the graphic functionalities of Microsoft's GDI+. This is the subsystem derived from the original GDI – one of the three core components for the user interface of Windows. GDI allows features such as drawing paths, rendering fonts and handling palettes. The new GDI+ version additionally features anti-aliasing capabilities, floating point coordinates, and support to output formats such as JPG or PNG. This is however a low-level library, adequate to do immediate rendering on a canvas. There is no support for interaction with the drawing, nor for anything related with graphs.

GDI+ provides services for three main categories: 2D vector graphics that can be built with primitives such as lines, arcs or bitmaps; imaging for non-vectorial

image needs (fast caching of bitmaps, for example); and typography for text related needs.

The System.Drawing library has a main class *Graphics* that represents a canvas where we can draw. Since it is an abstract class, we cannot instantiate one. We can however get a reference to an object's canvas and draw on it. Graphic objects can then be edited via the *Pen* class, the *Font* class or the *Brush* class. The first is used to draw lines with specified parameters such as width and line style. The second is used to format a piece of text via its font, size or appearance. The last one is used to define how a specific pen or font should be painted.

The System.Drawing library also provides many structures useful for programmers, such as the Rectangle, Circle, Point or Color structures. It also has the advantage of being easily portable to other OS, especially to Linux via the Mono framework [38].

2.6.2.2 Cairo

Cairo [39] is a 2D graphics library with support for multiple systems. Originally written in C for Linux, the actual version has several wrappers and is available for most output targets, such as PS, PDF, Windows, Linux or simple image buffers. Cairo also features hardware acceleration through *glitz*, and Mac OS support via *Quartz*. The original idea was to provide an adequate imaging model that could match modern application needs, using a common API that could be used consistently.

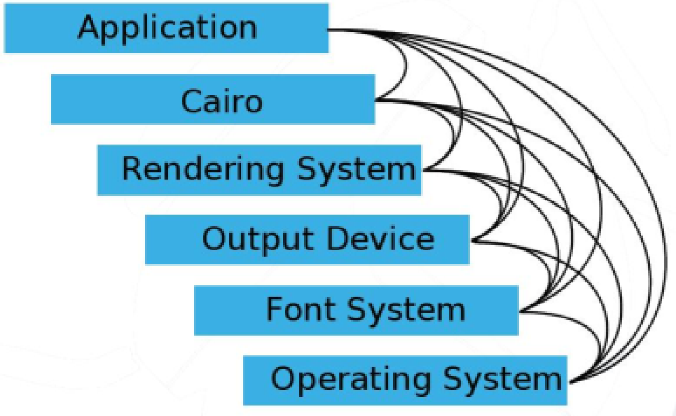


Figure 2.12: The Cairo architecture diagram.

The Cairo library is essentially a low-level one, dealing with direct rendering of graphics, text and fonts and rasterization. It is also *stateful*, meaning that it keeps track of the current path, coordinate system or color of the pen. Furthermore, the library doesn't provide any means of user interaction [40]. A rough schematic of

the system's architecture is shown in figure 2.12. A brief presentation on the Cairo drawing model and architecture can be found in [41].

Cairo's drawing model uses the concept of *destination*, *source*, *mask*, *path* and *context*. The destination represents the canvas, it can either be a PDF file or a computer screen. The source is the paint, with which we paint the canvas. This can be a color, a gradient, a set of colors or even a prerendered image. It also has transparency information. The mask acts as a stencil between the canvas and the paint. Anywhere allowed by the mask, the source will paint the destination, elsewhere it wont. The path is used to shape and model the mask into any desired form. Finally, the context is the information about all the previous concepts and some properties. This is then rendered using strokes, fillings or text characters.

Cairo is still being actively developed, and is used by some important names in the software industry – namely Inkscape, GTK+, Poppler and the Mozzila project are using it for diverse ends.

2.6.2.3 GraphViz

Whereas most of the previous libraries provide either high-level graphic interaction features directed specifically to graphs, or low-level graphic abstractions that cover not only graphs, GraphViz [42] is somewhere in the middle. It is a filtering library for graph drawing, taking text input from the user and producing a graph representation output. A in-depth look at this project is presented in [43].

The GraphViz Toolkit has two basic libraries: *Libgraph* and *Dynagraph*: the first implements attributed graph data structures and deals with input and output, and the second contains a set of incremental layout algorithms. The toolkit also has a set of graphical applications for specific functions, all built on top of these two libraries. The native language is C.

The problem with applying different filters or layouts to the same graph is that there isn't a standard graph description language, and that it is hard to create one that encompasses the needs of a wide range of algorithms. For that purpose, the creators of the libraries have developed a simple language called *Dot* that uses a simple syntax to describe graphs. This language allows for the user to input valuable information regarding the graph, information that may or may not be important to specific algorithms. Different algorithm implementations should look for specific lines that are relevant and deal with them, otherwise the information should be discarded.

After providing a correct text input, the second step is to apply a specific layout algorithm to the graph. GraphViz is capable of applying the Kamada-Kawai algorithm [19], the Fruchterman-Reingold algorithm [20], the radial layout algorithm by

Graham Willis [44] for very large graphs, the circular layout algorithm by Six and Tollis [45], and it also has its own hierarchical algorithm for directed graphs.

2.7 Graph Interaction

In this last section of literature review, we will make a brief overview on graph interaction, and on its importance on defining a positive working experience with graphs. Graph interaction refers to the way human-computer interaction is developed when it comes to graphs. It is the interface to the user, the tools we give the user to model and view the information we are displaying. Different programs have different levels of interaction: some allow for the nodes in a graph to be moved and others don't, for example. This level of interaction will guide the user to understanding what he can and can't do. The interaction model will determine whether all the hard work put underneath the interface was worth it or not. We may have the best algorithms and the best solution, but if all of that is topped off with a bad interface, all our hard work will be going down the drain. Users will often use our system in ways we never even though possible, and do mistakes they didn't expect to happen. Hence, the importance of a carefully thought out interface is invaluable. The book [46] provides a simple example of how even a carefully thought out interface can make experienced users make mistakes:

> "[...] The word-processing package we originally used to write this introduction is menu based. Menu items are grouped to reflect their function. The 'save' and 'delete' options, both of which are correctly classified as file-level operations, are consequently adjacent items in the menu. With a cursor controlled by a trackball it is all too easy for the hand to slip, inadvertently selecting delete instead of save. Of course, the delete option, being well thought out, pops up a confirmation box allowing the user to cancel a mistaken command. Unfortunately, the save option produces a very similar confirmation box – it was only as we hit the 'Confirm' button that we noticed the word 'delete' at the top..." [46].

Even simple well-thought out interfaces can make a nice interface look awful in the user's eyes. It is therefore important to think about the interface since the very beginning of the project, to iterate certain functionalities as the program is developed, to make it more concise and effective. As said before, [46] provides a nice introduction to the area of human-computer interaction. For the interested reader, we also suggest [47] for an academic overview of the field, and [48], which is considered one of the classic books about human-computer interaction.

2.7.1 Graph Oriented Interfaces

The next subsections will describe some interesting interfaces for programs dealing with graphs and with information visualization in general.

2.7.1.1 MusicMap

MusicMap is an ongoing project developed by DimVision [49] that presents information about albums or bands through a graph on a web browser. The idea is to allow the user to create and navigate a map of information to find out similar bands or albums after a specific search. The interface is minimalist, yet strives to put the control in the user's hand as much as possible. The input should be a search quote. The applet then returns a set of possible results, and the user should choose one start a new search. After one result is chosen, it will appear as a single node on the screen, with an image and a text caption.

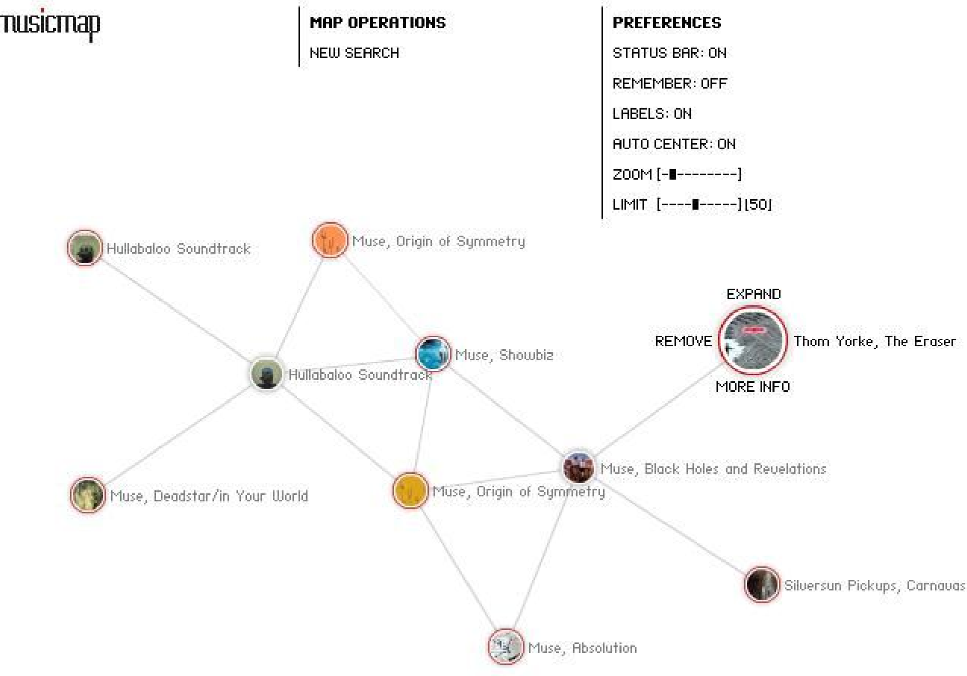


Figure 2.13: The MusicMap interface working.

Navigation is then achieved by mousing over a node. The node will grow a bit bigger, and present the user with choices to *expand* (show adjacent nodes), *remove* (remove the node from the graph) or *more info* (obtain info about the album or band). Presenting the choices near the node prevents the user from having to move the mouse around to reach menus. As the network grows, the graph will always try

to balance itself out. Users can still drag the nodes around, but as soon as they release it, the whole graph will slowly balance itself out again.

There is also a preference bar that allows the user to limit the number of nodes present at any one time, or zoom in or out on the graph area. The most impressive feature of this project is its simple yet intuitive interface. Instead of guiding the user, the applet encourages exploration, and allows the user to find out by itself how to use the features. Simplicity also makes the learning curve really smooth. How to navigate the graph is learned in a matter of minutes. The information showed is also kept to a minimum, avoiding unnecessary saturation of the screen.

2.7.1.2 The Opte Project

The Opte Project is a finished work started by Barret Lyon as a bet between friends. Back in 2003, Barret said he could map the entire Internet in a single day. His friends wanted to see the deed, and so this project was born [50]. The result is an open-source program that scans the net (or any other large scale network, for that matter), and produces a visual graph of said network.

Since the program has not visible interface, one might wonder why the reference here. We list this project because of two reasons: firstly, its outstanding work with very large graphs - some of the images produced have over 5 million edges and over 50 million hop counts; secondly, because this project shows how even graphs of this dimension can be drawn in an tractive and readable way. Of course that the idea here is not to navigate and read each node individually. But this project shows us that graphs can also be used to produce a visual representation of metaphysical universes, such as the Internet.

2.7.1.3 Websites as Graphs

This project [51] consists of a small applet that looks at the DOM[3] from any HTML page and converts its tags into a colored graph. Different tags such as divs, anchors, images or tables are given different colors. Since HTML is basically a set of nested tags, this graph is more or less a tree (there are no cycles), however the result is treated as a simple undirected graph. The interaction is very simple. The user simply inserts the URL for the page he wants to see as a graph and presses a button. The curious part is that the actual building and laying of the graph is done in real time, as the file is read. As the graph grows and distributes itself in the canvas, we get a visual representation of the structure of the site, its complexity, what type of tags were used and what were not.

[3]DOM stands for Document Object Model.

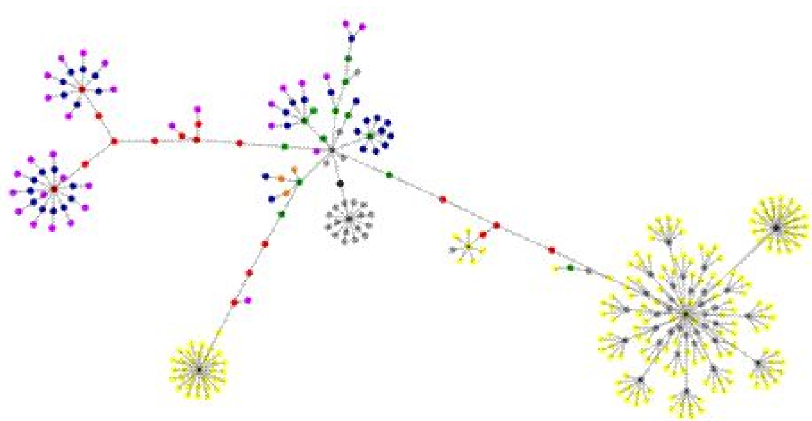


Figure 2.14: Representation of the apple.com site as a graph.

Although interaction is sparse in this applet, we decided to mention it because of its outstanding layout algorithm, which runs smoothly while the page is parsed and the graph is constructed. No indication is given, but we believe it to be a dynamic version of the Fruchterman-Reingold algorithm.

2.7.1.4 TouchGraph

The TouchGraph application is being developed by LLC [52] and is sold as an interface for exploring information from ever growing data collections. At their official website, the company presents the users with two free implementations of the tool for anyone to experience. One said implementation is the TouchGraph Google, that allows the results of a search to be presented as graphs of interconnected information, as presented in Figure 2.15.

The applet asks for a search as input and then constructs the graph using a force-based algorithm. This algorithm may be paused at any moment, allowing the user to drag nodes individually to obtain more visibility on a specific area. Different concepts for the same search, or keywords that may mean different things in different domains are clustered by color, allowing the user to quickly have a general idea of what he's searching for. The applet also allows users to specify a number of clusters, and the program will recalculate and color the nodes according to the level of detail specified. Colors can also be changed individually. Graphs can also be zoomed in or out, and edge length can be adjusted according to the density of the results.

To the left of the graph area, there is a list of the sites presented in the graph. This list can be filtered to show only one of the clusters, or only sites that would also appear on a second search. Finally, the sites' meta information will show up when they are selected via mouse click, together with a link to said site.

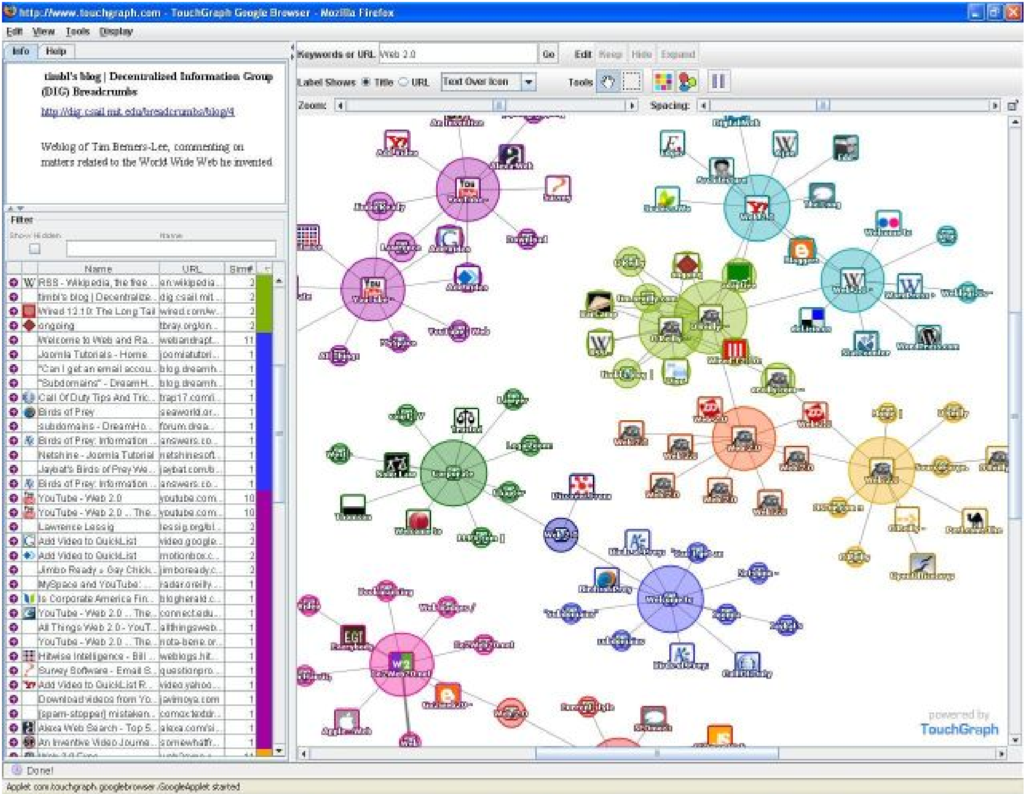


Figure 2.15: The TouchGraph interface for the search *web 2.0*.

With its many tools and features, TouchGraph is by far the most interactive interface we found in graph-related applets. The interface is not always intuitive, and the graph layout can sometimes get sluggish, specially if there is a lot of information on the screen. However, the sheer number of features and the ability to represent large quantities of information on the screen makes it a good example of graph interaction.

2.7.1.5 Visual Thesaurus

Visual Thesaurus is an on line thesaurus with over 145000 words to be explored using a graph. In the words of the official site [53],

> "It's a tool for people who think visually."

Basic input is a search for a specific word. A graph with the search term and related terms will be presented. The interesting concept here is that not only terms, but also meaning will be represented. Rolling over a node will make a small pop up show up, with the meaning of the word chosen. Green nodes represent verbs, violet nodes adverbs, yellow ones are for adjectives, and red nodes are nouns. Any of these groups can be turned on and off, to ensure that the exact word is found.

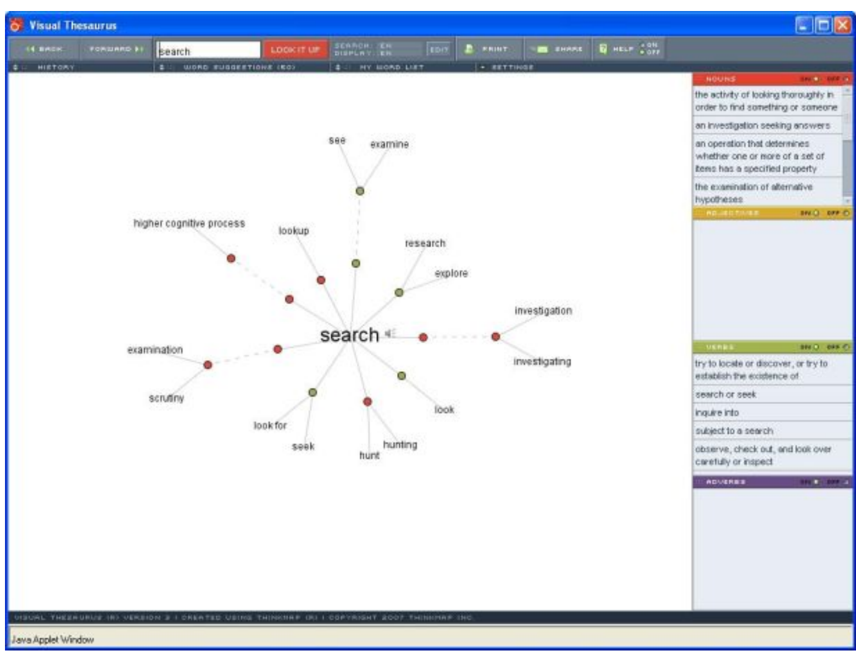


Figure 2.16: The Visual Thesaurus interface.

Users can navigate the map at leisure. Nodes can be dragged, but they will return to their original places. The layout is mostly circular, and once you double click a word to navigate to it, that word becomes the center of the graph and a new graph is designed. This guarantees that the screen will never be saturated. A sound for most words is also available. An example of this interface is shown in Figure 2.16:

Visual Thesaurus distinguishes itself from other applications using graphs because it is one specifically tailored for a thesaurus. Different types of relations are represented in different ways: an orange dashed line means that a word is *also related to*, while a gray dashed line means *is a type of*. The navigation is also more focused on the concepts of the thesaurus than on the visual representation of it.

2.7.2 Visualization Tools

The next subsections will describe some interesting interfaces for programs dealing with graphs and with information visualization in general.

2.7.2.1 We Feel Fine

We Feel Fine [54] is described by its authors as a feeling collector. What this applet does is collecting sentences from blogs all around the world every ten minutes. Each sentence with the expression *i feel* is saved and the associated feeling is marked

down. Using the blog's meta-information, it is sometimes possible to save additional information about the location, gender or age of the person who wrote the post, or the weather when he/she did so. All this information is saved in a database. The applet then lets anyone query that database in different ways.

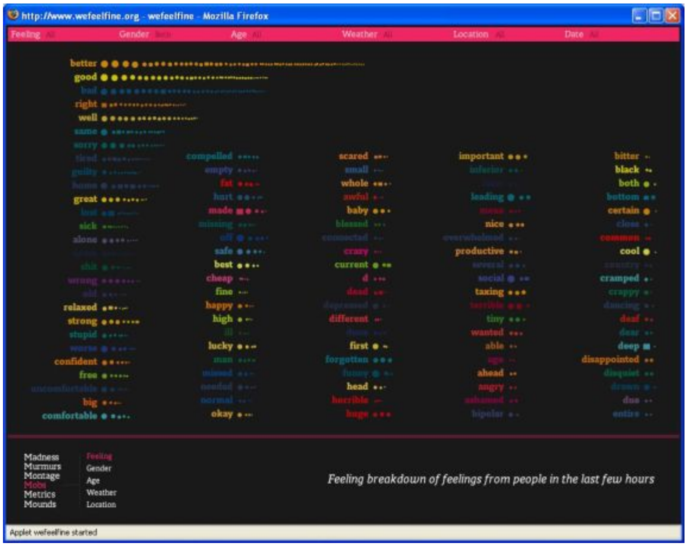


Figure 2.17: The We Feel Fine applet.

We Feel Fine give the users 6 *movements* to browse the information. It also provides a filter to choose a specific population. Users can choose to see only one feeling, or feelings from a certain age range or from a certain country. The feelings that satisfy the filter are then shown in any of the movements as simple dots colored according to the feeling they represent. One movement, for example, allows the users to see how the current population is divided in terms of statistics as gender, feeling, or date of post. Another lets the user extrapolate significant deviations between the current population and the average one. This is all done with a particle system in which each particle represents a feeling and its associated sentence. The particles will organize themselves according to the selected movement.

This applet is mentioned here because of its easy to use interface and intuitive features. Even though it doesn't use graphs, its interface could be used as a hint as to what a good interaction system could look like.

2.7.2.2 Web Trend Map 2008

The last project we would like to refer is the Web Trend Map 2008 [55]. Not really an applet, nor anything related to graphs, it is nevertheless an impressive example on how to visually present large amounts of information in a readable way. Created by a design team from Japan called Information Architects, this work uses the central

metropolitan Tokyo metro as a foundation to lay down the 300 trendiest sites in the Internet, according to their importance and popularity. The quantity of information is a big factor in determining how to represent it, and we feel that this work managed to present it in an intuitive and at the same time meaningful way. Figure 2.18 shows a part of the final work.

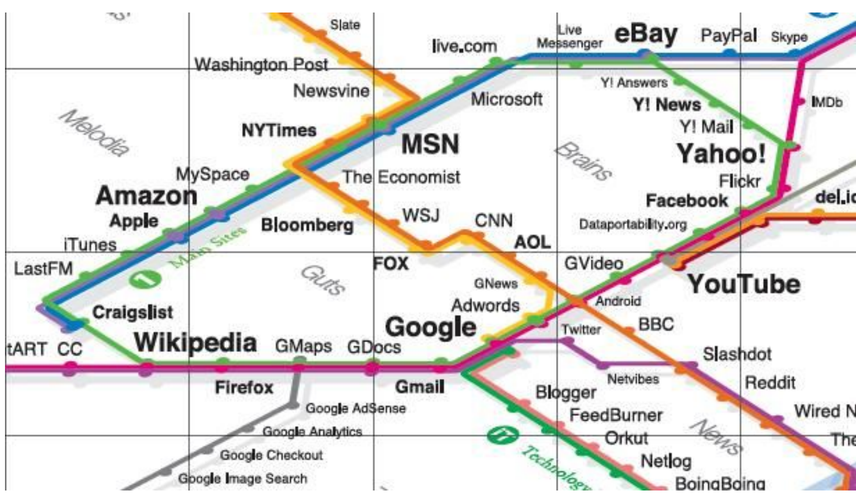


Figure 2.18: The Web Trend Map 2008 example.

Chapter 3

Methodology

After all the analysis made in terms of state of the art, it was time to define and decide how the graph's representation and interaction problem could be solved. In this chapter, a methodology is defined to create a graph interaction control (henceforth known as GIC). This involves a list of objectives for the project, the definition of the list of use cases and an in-depth study of several architectural details and choices made.

3.1 Problem Description and Objectives

As described in sections 1.2 and 1.3, the current project consists in the development of a customizable control that allows representation and interaction with graphs. These two fronts should focus on two specific cases of graphs: thesaurus representation and business organograms. By representation we mean a visual perspective of a graph's structure, one that helps users understand the bigger picture on a complex network of information, for example; by interaction we mean a simple and reliable way to navigate, edit and layout the graph, effectively simplifying the task of keeping the information up to date.

It was also stated by ParadigmaXis that the final result should be a reusable control, to be integrated with the GISA system to allow simple thesaurus navigation and editing. The only ground rule established by the company was that the solution should be coded in C#, and that it should be multi-platform. The following is a complete list of objectives set for the solution:

- Basic Features:
  - Graph Loading: the control should be able to load and read graph models either passed by the parent application or from a file;

- Graph Creation: the control should allow then creation of nodes, edges, and structural information of graphs on the control area;
- Graph Editing: the control should provide tools to edit node and edge related information, visual details (i.e, colors) as well as the graph aspect and layout (i.e, dragging nodes around);
- Graph Navigation: the control should provide a mode for graphs to be visited like a road map, having one selected node and showing the connected ones around it.
- Graph Layout: the control should provide a few layout algorithms to enhance user experience and present the graphs in a aesthetically pleasant form. Thesaurus should be presented in a grid-like fashion, and organograms in a directed, as planar as possible way;
- Style sheets: the control should be able to load and use style sheets created by users. These style sheets determine the visual appearance of every graph element on the control;
- Organograms: a different derived control should contain a slide bar at the bottom, allowing users to define a specific time interval and show the graph nodes pertaining to that interval.

• Interaction Paradigm:

- Standard-compliant: visual representation of graphs should follow as much as possible the rules defined in the ISO 2778 (Establishment and Development of Monolingual Thesauri) for thesaurus graphs; User-friendly: controls and interaction with graphs should be as simple as possible. All actions must provide visual clues so the user understands what is happening.
- Customizable: The depth of graph drawing when navigating and other information should be customizable by the user;
- Event-driven: The control should raise events accordingly to the actions performed on graph elements (i.e, creating or editing a node).

• Graphical Library:

- Open-source: the library should be open-source, preferably one with BSD licensing;
- Community: the library should have a large community and presence on the Internet, to allow for easy solving of problems;

  - Performance: the library should provide good to excellent performance in terms of screen painting. At least basic support for anti aliasing and double buffering should be provided;
  - Multi-platforming: the ability to port the code built with the graphic library easily to other operative systems such as Linux is highly valued.
  - Graphic Acceleration: the use of graphic board capabilities would be an advantage to the project.

- Architecture:
  - Modularity: different concepts and data models should be kept independent and the user should have access only to what is strictly necessary for the control to work;
  - Scalability: all data models should be expansible and easily changed to accommodate new concepts and properties;
  - Maintainability: classes and interfaces should be built in a way that small changes won't require much refactoring. Reusability is a big plus.
- C# Language: The control should be implemented in C#.
- Multi-platforming: The whole control should be, as much as possible, usable in other operative systems than not its native one.
- Unit Testing: the use of unit tests on all major classes is encouraged.

The above list is by no means complete. Graph representation and interaction using a control could go much further, but time constraints must also be taken into account. This project's focus was on proving that such a type of control could be developed within the available timeframe (20 weeks), and as such the list presents only a basic set of features desirable for the GIC.

Furthermore, some sort of priority had to be established between the goals set. It was accepted that unit testing, graphic acceleration and multi-platforming were considered secondary objectives.

Some concerns were also raised regarding two questions: firstly, how can the control follow the rules set by the ISO 2778 in a virtual environment? This ISO is quite old, and refers nothing about computer interfaces, so the challenge here is to follow the ISO as much as possible, while still maintaining fluid and user-friendly interfaces; secondly, how can the control provide such user-friendly interfaces? Human-computer interaction is a huge research area, and time does not allow for an in-depth analysis of many works to develop the *perfect* interface. For such a project, one can also set the goal of finding the right balance between instinct and theory on user interfaces.

3.2 Use Cases

A list of use cases is defined to determine exactly what types of tasks a specific application should provide, and what functionalities are present for each of the actors involved with the system. An actor is a final user of the system, and a use case is a feature of the system involving one or more actors. A use case diagram presents a graphic overview of this information in a manner such that every actor is easily identifiable and each use case is clearly connected to its associated actor.

For the GIC, we can define two types of actors:

- **User**: A general user that will interact with the control implemented in GISA or other application. This user has access to the two control modes: navigate and edit, and can perform most tasks regarding graphs, such as editing or applying a layout.
- **Programmer**: Represents a programmer implementing the control in an application. The programmer has control over some parameters that can be tuned in for specific cases, such as what type of tasks are allowed to users, or what are the default values for style sheets.

For the specific purpose of loading an existing graph on the control, we consider a programmer the same as an user, since loading methods are essentially the same for both actors. We can also subdivide the control in two specific sub-systems:

- **Navigation Mode**: This subsystem of use cases occurs when a graph is loaded to the control and is simply being navigated upon.
- **Edition Mode**: This applies to use cases when CRUD[1] occurs on a graph element.

Figure 3.1 presents the use case diagram for the GIC. The following list details each use case and what it refers to, grouping them according to their respective subsystem:

- Navigation Mode
  - Check Node: refers to the act of selecting a node from a graph and checking information about it, such as its name or type;
  - Check Edge: refers to the act of selecting an edge from the graph and reading information about it;

[1]CRUD stands for *create, read, update and delete"*

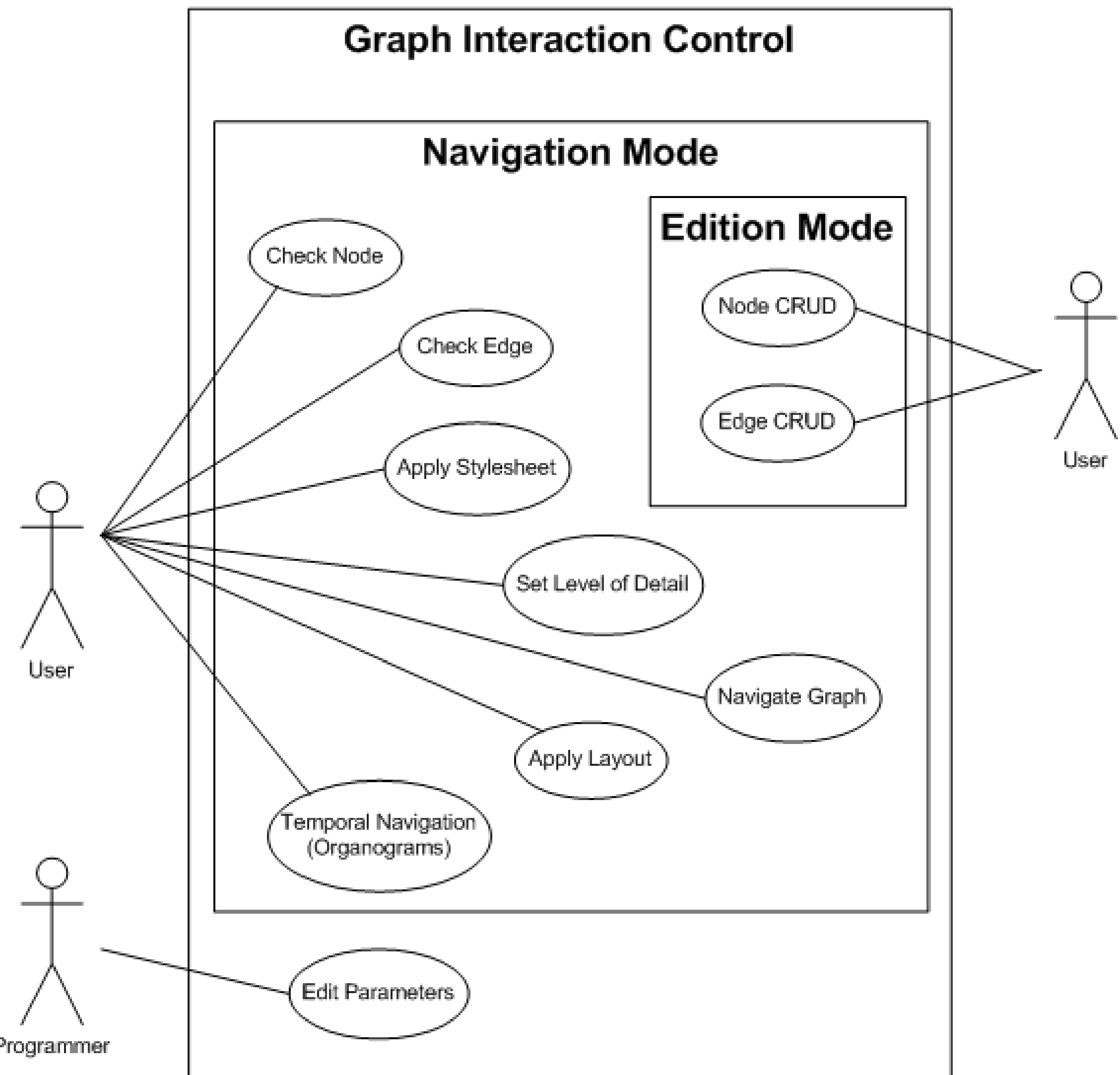


Figure 3.1: Use cases diagram.

– Apply Style Sheet: loading a new or different style sheet into the control, to change the appearance;

– Set Level of Detail: set the level of depth of drawing for graphs. When navigating on a large graph, this limits the number of nodes on the screen to avoid cluttering;

– Navigate Graph: the act of selecting a different node and concurrently centering the graph drawing around it.

– Apply Layout: applying a different layout to the current graph;

– Temporal Navigation (Organograms): using a slide bar on a control, the user can control the temporal interval to be shown taking into account the time stamp for each node, when presenting an organogram.

- Editing Mode
  - Node CRUD: The act of creating, reading, updating data about a node, or deleting it from a graph;
  - Edge CRUD: The act of creating, reading, updating data about an edge or deleting it from a graph.
- Other
  - Edit Parameters: Setting default values for the control, and determining its behavior via code, to specify it

3.3 Solution Architecture

With all the objectives set for the project, the design of the architecture for the solution was the next milestone. Finding a methodology for solving the graph interaction problem was no easy task: there were several decisions to be made and justified. In the following subsections, all the reasoning and decisions are presented: the decision of the data model to use in subsection 3.3.1, the choice of layout algorithms in subsection 3.3.2, the tests made to decide which graphical library to use in subsection 3.3.3 and the definition of the interaction paradigm in subsection 3.3.4. Finally, an in-depth look of the architecture of the control will be given in subsection 3.3.5.

3.3.1 Data Models Discussion

The data model is an abstract representation of how information should be accessed and treated by an application. For this project, it holds special importance, as it will be the main link between the GIC and GISA: the application will build or load instances of the data model and pass them to the control, who in turn will create an internal visual representation and keep a correspondence between each of the graph objects. This will keep internal control information and graph representations modular and independent from each other.

As presented in section 2.4, two models have been considered for representing graphs: incidence lists and incidence matrices. The first uses a list of nodes, each node having a list of related edges. Edges also keep information about each of the nodes they are attached to. The second uses a matrix of dimension $|V|^2$ and lists edges by filling matrix cells accordingly.

Three differentiating parameters arose while studying the above mentioned models:

- **Graph density**: incidence lists work better with low density graphs, incidence matrices prove better with denser graphs;
- **Data access speed**: checking a node's edges takes less time in an incidence list than in a matrix, but checking if an edge between two nodes exists is the exact opposite;
- **Redimensioning**: adding a node to an incidence matrix forces a new matrix to be draw, while adding an edge doesn't. The incidence list should be built on top of dynamic structures, otherwise each object added will force a redimensioning too.

A brief comparative study of the computational cost of several tasks on both of these models was undertaken to help decide which would be the most adequate for our project. Table 3.1 presents the results of that study, for a graph $G = (V, E)$, where $|V|$ is the number of nodes, $|E|$ is the number of edges, and v, u are nodes of G.

Table 3.1: Comparative study between incidence lists and matrixes.

Task	**List**	**Matrix**	**Best Choice**
Does edge (v, u) exists?	$\Theta(\|E\|)$	$\Theta(1)$	Matrix
What are the edges connecting to v?	$\Theta(1)$	$\Theta(\|V\|)$	List
How long to check all the graph?	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|^2)$	List
How long to create an edge?	$\Theta(3)$	$\Theta(1)$	Matrix
Total memory space needed.	$\|V\| + \|E\|$	$\|V\|^2$	List, if graph density is low

Checking if an edge (v, u) exists is immediate on a matrix, since the result is on cell $M[v, u]$. However in an incidence list it forces a run through the edge list, taking up to $|E|$ iterations to check if the edge doesn't exist. On the other hand, checking the edges starting or ending on a node is immediate on the list, since each node contains a list of edges associated. On a matrix, this takes $|V|$ iterations, since a full row of the matrix must be checked. Running through the whole graph is faster on the list as long as $|V| + |E| < |V|^2$, but since the control will work with sparse graphs, we can assume that lists are a better choice here. Creating or deleting edges is quite fast on matrices, since its a matter of changing the value of a cell; on a list, this can take up to three times longer, since one must add the new edge to the edge list, and to each of the nodes' edge list that the new edge connects. Finally, the total memory space needed is also lower in an incidence list as long as $|V| + |E| < |V|^2$. Once again, since sparse graphs are what will be loaded into the control, the incidence list is considered a better option.

After this brief study, two things must be considered before making a decision: that the GIC will be essentially working with low density graphs, and that dynamic editing of the graph is one of the main objectives for the control. This leads to the conclusion that the incidence list the most adequate choice.

According to [13], a simple and effective implementation of an incidence list can be made with three classes: Graph, Node and Edge. A Graph object contains a list of nodes and a list of edges. A Node object contains information about a node (name, value, type), and a list of pointers to Edge objects associated with itself. An Edge object contains information about an edge (directionality, label, type) and a pointer to its start and end nodes. Each object also has an unique identifier.

Figure 3.2 presents a class diagram for this implementation:

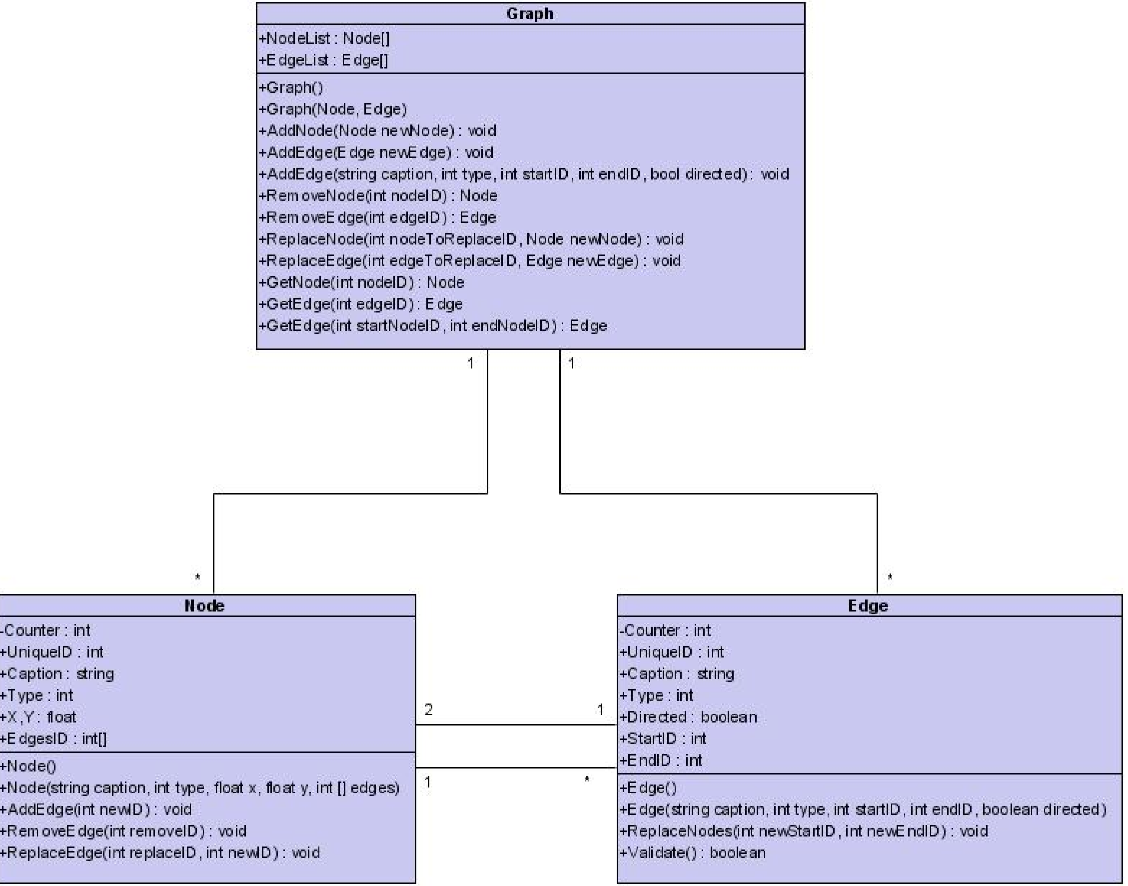


Figure 3.2: Class diagram for the incidence list.

Further details on the implementation of this data model can be found in section 4.1.

3.3.2 Layout Algorithms Discussion

The layout algorithm will be responsible for the aesthetic look of any graph loaded on the control. The choice of an appropriate algorithm is important, since one of the main goals of the project is to provide a visual and accurate representation of a graph. For the GIC, several straight-line algorithms were considered, both for the thesaurus and the organogram case. In section 2.5 a description of the following layout algorithms was given:

- **Force Directed Spring Layout**[18]: edges are simulated as springs and a system with minimum energy is calculated. It was one of the first straight-line algorithms for graph layout;

- **Kamada-Kawai**[19]: an improved version of the previous algorithm, this algorithm rectifies some of the formulas used by Eades and presents some optimizations – it can be extended to layout hierarchical graphs;

- **Fruchterman-Reingold**[20]: nodes act as small particles, repelling each other the closer they are, and drawing closer if there's an edge connecting them. There is also an improved version of the algorithm, that optimizes run times;

- **Sugiyama-Misue**[15]: this algorithm uses four different heuristics to solve and present a graph as planar, whenever possible. The trade out is computational cost. The algorithm was made specifically for hierarchical graphs;

- **Simmulated Annealing**[27]: simulating a physical phenomena, this algorithm makes a high number of small variations on a system and checks if the total energy of the system is lower, over a number of iterations. Displacement is limited by a thermometer, that gradually cools the system down.

The first three algorithms are force based, and simulate a system of particles; the last two use different heuristics to find the best possible system from a set of possibilities. We can compare the advantages and disadvantages of both kinds of algorithms:

- **Force-based**: Simple to implement, with low computational costs, and hence suited for dynamic graph layout. Results are fast and aesthetically pleasant, provided the graph isn't too dense. They tend to reveal symmetries and to distribute nodes evenly across the drawing area. However, planarity is not ensured, even with planar graphs. Optimization isn't the best, since different number of iterations and initial layouts result in very different results;

- **Heuristically oriented**: Optimal results can be found, since the value of different aesthetic criteria can be varied to find the best solution. Planarity is normally achieved, provided the graph is planar. These algorithms are harder to implement, and have a high computational cost. Not suited for dynamic graph layout.

Table 3.2: Comparative study of layout algorithms.

Algorithm	**Iteration Cost**	**Iterations Suggested**	**Other Details**
Spring Layout	$\Theta(\|V\|^3)$	100	n\a
Kamada-Kawai	$\Theta(\|V\|^3)$	Depends on iterations t of inner loop.	Can be extended to hierarchical graphs.
Fruchterman-Reingold	$\Theta(\|E\| + \|V\|^2)$	Can be made function of $\|V\|$ or $\|E\|$.	Has improved version with iteration cost $\Theta(\|E\| + \|V\|)$
Sugiyama-Misue	$\Theta(\|V\|^2 \times 3 + \Theta(t\|V\|))$	Four phase algorithm with one pass. Phase 3 iterates t times.	Works only on hierarchical graphs. Composed of three NP-hard problems and one NP-complete.
Simulated Annealing	$\|V\|^2 \times \|E\|$	$30\|V\|$	Uses temperature concept, found by experimentation.

Table 3.2 presents a brief comparative study of the computational costs and details of each algorithm. Together with the layout algorithms, section 2.5 presented several aesthetic criteria, describing metrics that help enhance the visual aspect of a graph. These include planarity (no edge-crossing), symmetry, straight-line drawing, even distribution of nodes in the drawing area, and directionality (directed edges should point in the same general direction). It was also said that it is most of the times impossible to conjugate all these criteria, so it becomes important to define what will be considered more and less important.

Planarity helps a user perceive a graph more logically and with structure, should it exist. It is however a hard criterion to achieve, as not all graphs are planar. Symmetry and even node distribution, when present, tend to add structure to a graph, and to simplify the understanding of any subjacent structure. Most of the algorithms studied simulate particles or physical systems' behavior, and both normally tend to distribute objects evenly, showing symmetric patterns if they exist. It could be argued that these two criteria are easier to achieve given the proposed

algorithms. Straight line drawing adds simplicity to the graph, and saves the user the work from following complex edges between two nodes. Since most algorithms studied are straight-line oriented (except for Sugiyama-Misue, who allows for bends when necessary), this is an easy to achieve metric. Directionality mostly applies to organograms, where a specific hierarchy flow must be present. Of all the proposed algorithms, Sugiyama-Misue is the only one that deals with this criterion, although Kamada-Kawai can be extended to draw directed graphs as long as nodes are separated in a level-wise manner [17].

The GIC will deal essentially with two kinds of graphs: thesauri graphs, who should present themselves in a structured way, and organograms, who should be shown as a directed and planar set of entities. This means each case needs a different treatment, and that at least two algorithms need to be chosen for graph layout. Since interaction is the fulcrum of the control, fast and effective algorithms must be chosen.

For thesaurus representation, all force based algorithms and simulated annealing could be considered. However, Fruchterman-Reingold seems to be the appropriate choice. It has the lowest iteration cost for force based algorithms, provides an optimization model to reduce the iteration cost, shows existing symmetries, reduces edge crossing, while still distributing nodes evenly across the control. It is also trivial to implement, contrary to simulated annealing.

For organograms, two options arise: Sugiyama Misue and Kamada-Kawai. While Sugiyama's solution seems appropriate for hierarchical graphs, it is also true that its computational cost is high, and that implementing such an algorithm would be complicated. Kamada-Kawai provides a similar solution with a much simpler implementation and a simple variation to achieve the desired result. Sugiyama's algorithm is also not optimized for dynamic graph layouts, which makes Kamada the best option for organograms.

3.3.3 Graphic Library Choice

The choice of the graphic library is an important one, as it will define greatly the architecture of the GIC. The decision is twofold: firstly, the level of abstraction a library provides will influence the work that needs to be developed: a low level library will give the developers more control over the drawing tools and force the development from scratch of all interaction and layout task, whereas a high abstraction framework will probably provide some of those functionalities inherently at the cost of less customization power. Secondly, details like the license type, the ability to port code to other operative systems that not the native one or the languages it provides support for are all factors that need to be taken into consideration.

Table 3.3: Brief comparison of graphic libraries.

Library	License	Portable	Acceleration	Abstraction	Languages
System.Drawing	Free	Yes	No	Low	C#
Cairo	LGPL	Yes	Yes	Low	C, C^{++}, C#
Piccolo	BSD	No	No	High	C#, Java
Netron	GPL	No	No	High	C^{++}, C#
JUNG	BSD	Yes	No	High	Java
prefuse	BSD	Yes	No	High	Java

During the course of the investigation, several libraries were considered and studied, as presented in section 2.6:

- **System.Drawing**: part of the .NET Framework, this namespace provides access to GDI+ basic functionalities. There is a framework called Mono that covers System.Drawing making it portable.
- **Cairo**[39]: 2D graphic library with support for multiple output devices, including graphic acceleration through the X Render extension. Several wrappers exist for different languages;
- **Piccolo**[29]: toolkit that supports development of 2D structured graphic applets, with some focus on zoomable and pannable projects. Uses a scenegraph to describe objects on the screen and interact with them;
- **Netron**[31]: generic diagramming and graph layout library for the .NET. Provides users with a vast array of layout algorithms and a customized integrated development environment called Cobalt IDE;
- **JUNG**[33]: graph oriented framework for modeling and visualization of network data. Provides several algorithms for depth search, layout and filtering, built in Java;
- **Prefuse**[35]: library that provides extensive coverage of current visualization models in a large spectrum of areas, including graphs. Focuses on interaction and presentation of large sets of information.

The GraphViz [42] framework was also mentioned, but won't be considered for the solution, since it provides support for output devices not of our interest. Table 3.3 shows a comparison between the different libraries in terms of license, portability, graphic acceleration capabilities, level of abstraction and native languages:

The process of choosing one library for the control was divided into several steps. Several factors were taken into account, namely: the native language, the ability to

port the solution, the license type, the performance, the stability and finally the community activeness of the graphic library.

The GIC will, in the foreseeable future, integrate itself with the GISA application, built on C# language. This helped to limit the choice to those libraries that provide at least a wrapper for C#. This was not particularly promising, as both solutions that were only available in Java (JUNG and Prefuse) appeared to be well designed and oriented at graphs.

It was also stated by ParadigmaXis that, if possible, a free solution should be used. The license for each library was also a factor to ponder. The GPL license that Netron is under means that any derived work from the library would also have to release its code freely, hence Netron was also discarded as an option.

Piccolo also presented itself as a powerful tool. The license was acceptable and so was the language. Some test applications were run to verify the potential of the library. However, as questions and the need for some more information arose, it became clear that the community using Piccolo was not as active as desirable. Apart from a sparsely visited mailing list, there is virtually no support for using this library.

This narrowed the choice down to two candidates: Microsoft's own System.Drawing library, or Cairo. The latter appeared to have the upper hand, providing easy portability, a thriving community on the Internet, and the possibility to add graphic acceleration to the code. However, the C# wrapper for it provided a performance issue when not using graphical acceleration.

In order to compare the performance of System.Drawing and Cairo, a small implementation of the popular Tetris game was made with both libraries. The game was built in C#, with each version doing the drawing part using the respective graphic library. The game field was refreshed on a regular interval, with a preview of the next piece also being drawn when necessary. The pieces for the game were drawn using circular gradients in Cairo, and rectangular gradients in System.Drawing. The result was similar in terms of appearance, but not in terms of performance. Figure 3.3 shows a screenshot of both versions of the game:

Cairo's wrapper for C# worked well in Windows, but the image had to be composed in memory and then copied onto the canvas to work. This made the program a bit more sluggish than the System.Drawing counterpart, with some flickering showing up when the game board was refreshed. On the other hand, System.Drawing showed a much better performance, with no visible flickering even without double buffering. The game also proved to be cross-platform enabled through the use of Mono to port the graphic capabilities of the System.Drawing namespace on other OS.

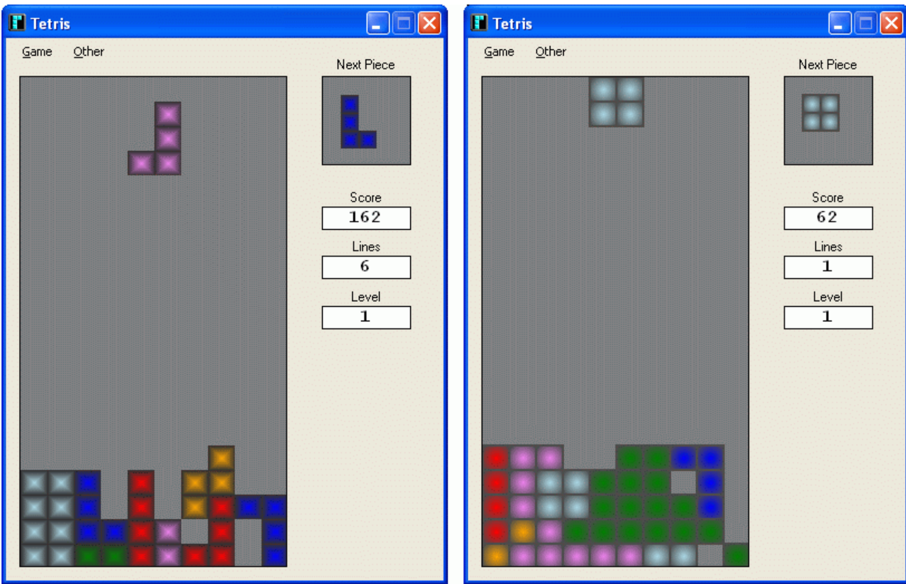


Figure 3.3: Screenshot comparison of tetris on System.Drawing (left) and Cairo (right).

In order to further compare the two versions of the program, a profiler was used to check times spent by both applications in common drawing tasks of the game. The test case was the same on both programs, corresponding to a period of 10 seconds of run time with a piece falling down on the game field. Table 3.4 shows the results of these tests, express in percentage of the total time spent doing the respective action, for a program that takes 100 units of time to complete:

Table 3.4: Time comparison between System.Drawing and Cairo.

Action	**System.Drawing**	**Cairo**
Erase Piece	0.10	0.93
Redraw Piece	0.03	1.10
Apply Gradient	0.58	1.53
New Game (Draw Field)	0.61	0.93

The profiling results were clear. In virtue of having to copy the image composed by Cairo onto the canvas for every drawing task, the library took generally more time to refresh the game area. This resulted in some visible flickering, as previously mentioned, and a clearly lower performance than the Microsoft counterpart. This lead to the choice of the System.Drawing library to develop the GIC: not only does it provide good performance and a good level of control over the drawing capabilities of the processor, it is also free to use and has a large active community on the Internet. Furthermore, System.Drawing's cross-platforming capabilities through Mono [38] perfectly fits one of the objectives set for the GIC: portability to Linux.

3.3.4 User Interface Principles

It is difficult to justify the choices made in this aspect, as the theory about user interfaces and usability issues is vast and many times contradictory or misleading. Although some works of major relevance in this field have been briefly studied ([46], [47] and [48]), the amount of information and analysis needed to decide the most adequate interaction process for this project would be tremendous. Therefore, the choices made for the GIC were mainly based on the team's personal experience developing user interfaces.

Since the control will deal with objects on the screen, an OOUI[2] is the obvious type of user interface to implement. This means that users should be able to perceive and act on objects, classify them based on how they behave, and maintain a coherent overall representation of the objects on the screen. A simple interface metaphor was thought out for this, similar to the famous desktop metaphor [56]. Instead of the desktop with elements of everyday work, this metaphor represents a table on which puzzle pieces are being placed. Each node is basically a piece of the puzzle, and each edge is represents the connection between two pieces. Users can then add new pieces or remove existing ones, move the pieces around to structure the puzzle and connect the pieces to add coherence to the graph. This takes advantages of the specific knowledge that users have of other domains, making the learning curve easier and the interaction process fairly faster.

On the next subsections, some ground rules are set regarding design principles, and an interaction paradigm is devised for the GIC.

3.3.4.1 Design Principles

Design principles are principles of human perception that can be utilized to create an effective user interface. They help reduce the number of errors, increase user efficiency and satisfaction, and smooth the learning curve. In [57], the authors mention thirteen main principles of display design, aimed at supporting human perception, situation awareness and high understandability. Some of these principles may seem contradictory, or even impossible to implement for specific cases. Indeed, there are no rules as to which should be considered more or less important Each case is a case, and these principles are meant to be tailored to best suit specific problems. For the GIC, the following have been selected as paramount to the success of the user interface:

- Legibility: good legibility is important to help users discern what is that they are interacting with. If nodes or edges cannot be correctly identified, then the control won't help the user to understand underlying structures;

[2] OOUI stands for Object Oriented User Interface.

- Redundancy gain: if a signal is presented several times, chances are it will be understood correctly. A selected node, for example, can be presented with a doubled outer shape, to help users understand its importance;
- Pictorial realism: objects should be represented as they are normally on real world. A father node is expected to be higher than its son, and organograms are expected to present its nodes as rectangular shapes, for example;
- Proximity compatibility: different sets of information about the same object should be linked by proximity. A node's label will be easily paired with the node if it is placed near it. However, an appropriate threshold of information must be found, as too many information sources too close may result in cluttering;
- Predictive aiding: helping the user with information that is not necessarily important but might reduce time spent performing tasks in the future helps the users consider future possibilities. Showing an edge's label when the mouse is near it reminds the user about information with potential value in the near future;
- Consistency: As the saying goes, old habits die hard. Long term memory from experience with other interfaces will arise, and users will expect the new interface to work in a consistent manner. Many users will expect a right click to raise a context menu, for example.

This means in no way that the remaining principles aren't important: they should all be taken into consideration when designing the interaction paradigm. It means only that these are mostly oriented to aspects related to the control being developed.

3.3.4.2 Interaction Paradigm

Interaction paradigm refers to the interaction process that users will have on the graph control. Some points were taken into considerations when deciding how interaction should work. First of all, the mouse should be the main tool for interacting with the control. Keyboards could (and should) be used as shortcuts to certain tasks, but tasks performed on a two-dimensional canvas needs to be both fast and precise. Interaction with a mouse provides this. It was also clear from the start that similar operations should be performed in similar ways. So the process of, for example, deleting a node or an edge should be essentially the same. This prevents the user from shifting to a different paradigm when editing different parts of a graph.

Finally, it seemed logic that each mouse button should process a set of tasks based on their type. Creating or moving a node are different types of actions. While creating involves drawing something on the screen that wasn't there before, moving

involves selecting an already existing object. Several ideas were tossed around the table and one came up that seemed to be quite logical:

- **Left mouse button**: Access to existing objects in the graph. Provides interaction to generic, non harmful tasks.
  - Node/edge selection: left mouse click over the desired node/edge selects it for editing or navigating.
  - Node moving: left mouse click over a node and dragging the mouse around will move the selected node until the mouse button is released;
- **Right mouse button**: Creation and deletion of graph objects. Provides interaction to specific tasks.
  - Node creation: right mouse click on an empty place shows a context menu with the option to create a new node. Alternatively, double click on an empty spot will do the same thing;
  - Node erasing: right mouse click over a node shows a context menu with the option to erase the selected node;
  - Edge creation: right mouse click over the start node and dragging the mouse over the end node will create an edge between the two selected nodes;
  - Edge erasing: right mouse click over an edge shows a context menu with the option to erase the selected edge;

Since the left mouse button is commonly considered the main mouse button, simple and unchanging tasks were assigned to it. This means that a new user won't accidentally change or delete potentially vital information by clicking around and trying out the control. This boosts the user's confidence by giving him a sense of ease of control while protecting him from unwanted changes.

The right mouse button, on the other hand, is assigned to important tasks that change the layout and structure of graphs such as creating new nodes or edges. Users tend to use the right mouse button for specific tasks and don't generally click this button unless they are looking for a way to perform those actions. This further encourages the user to feel in control. Also, all actions performed with the right mouse button will either make a context menu appear or require some dragging to complete. This will force the user to further prove what he intends to do, reinforcing safety against unwanted actions.

3.3.5 Architecture Description

Architectural patterns describes the fundamental structural organization schema for a software project. They presents the most basic subsystems of the software, their responsibilities and interrelations, and provide a broad representation of the project and all its encompassed parts. Some common architectural patterns include *peer-to-peer*, where each participant on a network is able to communicate to all others with no need for a centralized control point (such as a server); or the *service-oriented* architecture, where functionality is grouped around processes that can be described as services.

Despite the fact that different patterns focus on different software types, they are not by themselves an architecture. Instead they convey an image which captures the essential components of a system and how they relate to each other. In this chapter, justification for the use of the the model-view-controller (MVC for short) paradigm is given, explaining why it fits the restrictions and objectives for the GIC.

The MVC paradigm (as presented in [58] and [59]) is used with the intention to isolate each of the business logic components from user interface considerations. This means that a correct implementation of the MCV allows data models, data controllers and user interfaces to be essentially independent. Further changes in each of these subsystems are subsequently easier to perform, as they won't affect other parts of the software. Since view, model and controlling of both are decoupled, this pattern reduces the overall complexity of the system and adds flexibility and reusability. Figure 3.4 shows a rough diagram of the MCV paradigm.

- The **model** is the basic data representation for the domain of information of the software application. It gives basic meaning to raw data, and is possibly used by external applications to feed information into a program. In this sense, they are also responsible for exposing functionality and information;

- The **view** renders the information from the data model into a form suitable for interaction with users. It also requests the control to update the model, sending the controller information about user inputs;

- The **controller** takes care of event handling and user input through the information received from the view. This information is processed and mapped accordingly, and the necessary changes are invoked in the model. It basically controls the behavior of the application.

Objective wise, three quality factors concerning the architecture of the GIC were mentioned in section 3.1: modularity, scalability and maintainability. The MVC paradigm was our choice because it fits the aforementioned factors: modularity is

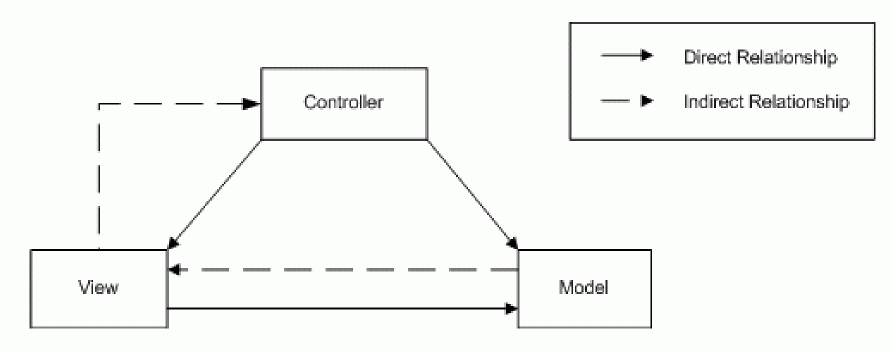


Figure 3.4: The model-view-controller diagram.

achieved because even though the controller, model and views work together for the application, they can be built independently. Scalability is also achieved, since new additions to any of the modules won't require refactoring of the remaining ones. The MVC paradigm also has a low maintenance cost, as debugging and updating can also be done just on the required parts of the software, leaving the rest untouched.

In the GIC case, the domain are graphs, so the model will represent information about them, its nodes and edges. The incidence list described in section 2.4 will be implemented with this objective in mind, so the model will consist of three classes, one for each of the basic components of the domain: Graph, Node and Edge.

The view will present to the user a graphical representation of a graph. The domain here focuses on graphical equivalents to nodes and edges. Most visual concepts for graphs revolve around simple shapes representing nodes and some sort of line connecting them. Therefore, a generalization for the view should include at least two basic concept: shapes and lines. Putting together these two objects, an accurate rendering of the information kept in the model can be constructed. The view should then contain the two basic classes depicted previously: Shape and Line.

The GIC itself will represent the controller, managing information in the incidence list and showing information to the users through the view. The control is responsible for raising events and for keeping a matching between the objects representing the view and the model. An application using the control will know only about the model, whereas the users will interact only with the view. It is up to the GIC to guarantee that any changes in one of the components will be correctly reflected on the other. The GIC is also in charge of keeping a style sheet and applying it correctly to the view of the graph.

One of the main goals of the MVC is to provide the decoupling of subsystems of an application. This allows future and independent use of at least some parts of it in other projects. For the GIC, it was established since the very beginning

that the data model should be reusable, or at the very least usable as a standalone library. The objective of this is simple: to allow any kind of software to build graph representations that could be used with the control.

Furthermore, complete independence between a graph representation and the visual representation of it is also important. This addresses security concerns such as the level of control users have over the graph information. Since interaction is mainly done with the visual representation, if both models are independent, changes can be checked and validated before the real representation is changed too. Figure 3.5 shows a scheme of how the system (codenamed Olympus) has been thought out to work:

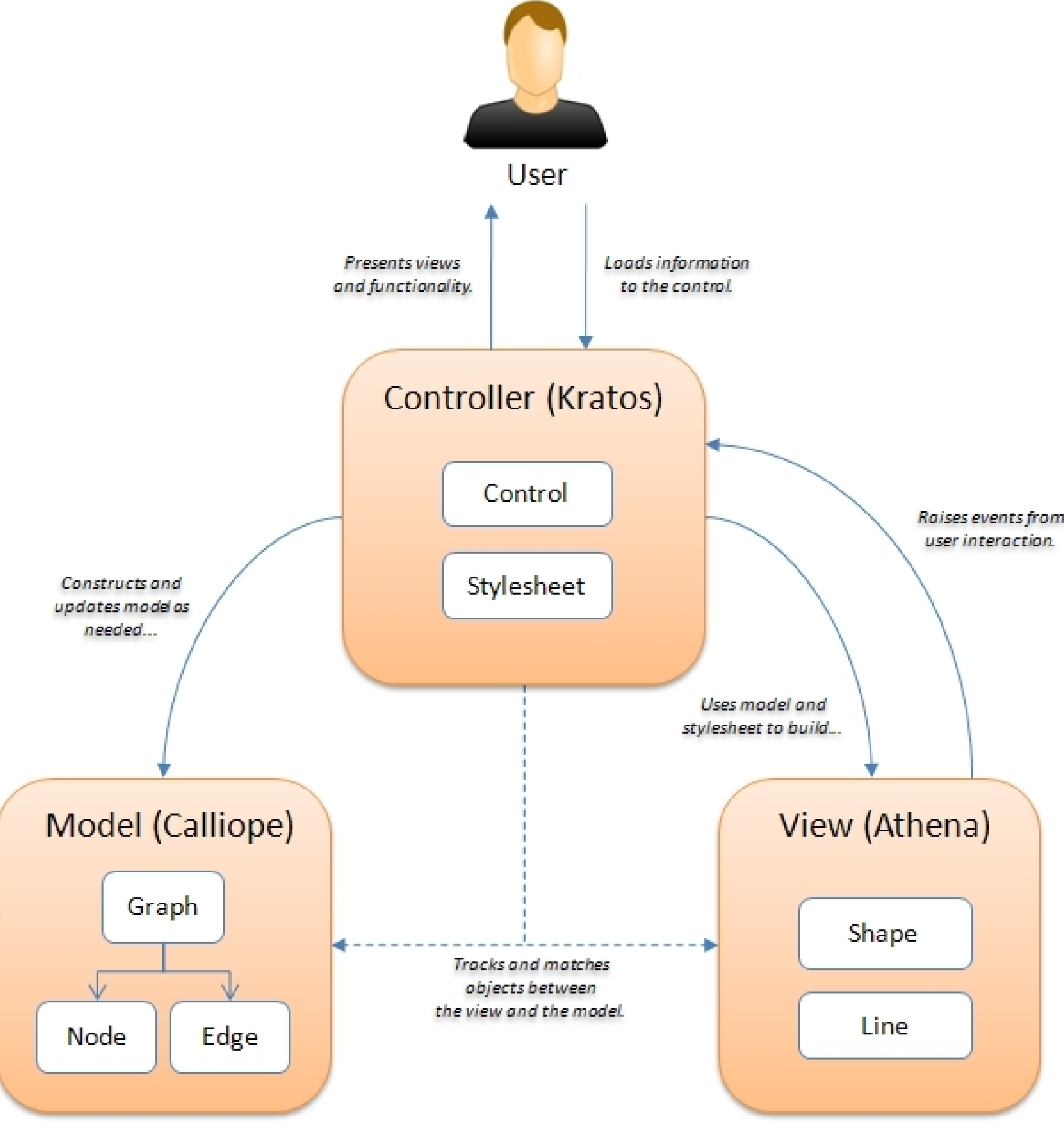


Figure 3.5: Scheme depicting the main classes to be used by the GIC.

The project has been thought out as a set of three separate libraries, one for each module of the MVC paradigm. The names given to each are all inspired in Greek mythology and reflect the purpose and intention of each part:

- **Calliope library**: This library is the data model implementation of an incidence list to represent Graph. It consists of the three basic concepts needed to represent a graph: graphs, nodes and edges;
- **Athena library**: This library represents the visual side of the project, and contains the basic elements needed to represent a graph on a screen: shapes and lines;
- **Kratos library**: This library contains the control itself, as well as the implementation of the style sheets.

The user interacts solely with the control and with a visual representation of a graph. Typically, a graph constructed using the model is loaded by the user into the control. The control then uses this information together with the information contained in the style sheet to build the view for that specific graph. The view is presented to the user, and raises events accordingly as the user interacts with it. These events are dealt by the controller, who is in charge of validating and changing the model and the view as requested by the user.

The controller should also keep an exact correspondence between the model and the view at all times. Each node in the graph should be paired with a shape, and each edge with a line. This way, if a shape is changed, the control also changes the corresponding node as needed. If a line is erased, the correspondent edge is also erased. If a new object is created, the corresponding object in the model should also be created.

Chapter 4

Implementation

In the previous chapter, the GIC's architecture, codenamed Olympus Library, was described as a set of three smaller libraries that represent the three components of the Model-View-Controller paradigm: the Calliope library corresponds to the model and provides an abstract representation of a graph and its information; the Athena library corresponds to the view and provides the controller with visual representations of graph objects on the screen; finally, the Kratos library represents the controller and contains not only the controller itself, as the means to keep information flowing between the two other libraries. This division is meant to allow reusability of any of the classes for other projects, as well as allowing GISA to work independently from the control. A prototype was also developed in the form of an independent application that serves as a showcase to the control's functionalities.

A description of the prototype constructed for the GIC is made in this chapter: each of the libraries is presented in greater detail, and the implementation of the prototype is explained. This provides the reader with a technological and low level insight into the process of developing the GIC, whereas the previous chapter focused on high-level architectural details and decisions. It also shows what was changed from the decisions taken in the previous chapters, and why.

Sections 4.1, 4.2 and 4.3 describe the Calliope, Athena and Kratos libraries in greater detail; section 4.4 focuses on the prototype itself and the functionalities it provides; finally, section 4.5 draws a closing line on the implementation of the project with some final words. Class diagrams will be presented as the example diagram presented in Appendix B.

4.1 The Calliope Library

The Calliope library is a direct implementation of the data model described in section 2.4.1, and provides an effective method for storing information about a graph – an incidence list. It is loosely based on the implementation suggested in [13], which suggests the use of three basic classes: class Node for representing nodes, class Edge for represented edges and a global class Graph that keeps track of the nodes and edges of a graph. A graph can have any number of nodes and edges associated to it. A Node has any number of edges associated to it too, but an edge can only (and *must*) have two nodes associated with it, as shown in Figure 4.1.

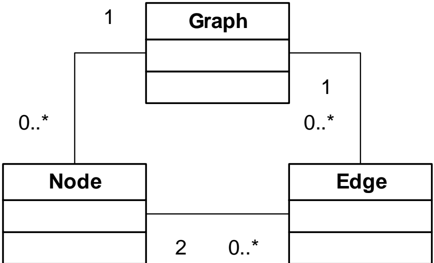


Figure 4.1: The Calliope class diagram.

The implementation of this library was pretty straightforward: there were no special functionalities to achieve, and no special cautions to take. The major implementation decision here was regarding element types. Both nodes and edges in a graph should be categorized by specific types: this allows not only for style sheets to be applied to specific types of elements, but also to categorize parts of thesaurus accordingly to their types, in conformity with GISA.

The original idea was to allow these types to be a dynamic list applied to elements and graphs. However, and as time passed and the GIC was implemented, this became a secondary concern and more of a roadmap feature[1] than a primary objective. The solution to this was to use simple enumerations in each class that contain the types of nodes and edges present in GISA. This allows for compatibility with GISA in the near future, and leaves room to turn the type concept into something more dynamic at some point in the future.

The next sections present each of the aforementioned classes in detail, explaining the information they contain and a summary of the functionalities they provide.

[1]A *road map* objective is one that is set for the far future.

4.1.1 The Node Class

The Node class represents instances of nodes of a given graph, in a conceptual form, as presented in Figure 4.2.

Node

-counter : int
-visited : bool
-level : int
+ID : int
+Caption : string
+Type : NodeType
+Edges : Edge

«C» -Node()
«C» +Node(in caption, in type, in edges)
+AddEdge(in edge : Graph) : void
+RemoveEdge(in edge : Graph) : void
+ReplaceEdge(in edge : Edge, in newEdge : Edge) : void
+Clear() : void
+Cardinality() : int

Figure 4.2: The Node class.

The ID property identifies an instance of the class as an unique member of a graph by incrementing the static int counter each time the class is instanced. The Caption property contains the node's text description, and is optional. The Type property identifies the generic type of node the instance represents. This type follows the NodeType enumeration of the same class that contains all the types of entities present in GISA: Form, Place, Name, Topic, and Creator. The Custom type was added for any other types of entities being represented. Eventually, the idea is to turn this property into something more dynamic, allowing for new types of nodes to be defined on the fly. Finally, the Edges array of edges represents the edges that either start or end at the instanced node.

The class provides two constructors: a private Node() that is barred to external users, and a generic Node(caption, type, edges) that builds a new node with a specified caption, type, and edge list. The edge list can be null.

The three AddEdge(edge), RemoveEdge(edge) and ReplaceEdge(edge,newEdge) methods are self-explanatory, and fill the information about edges connected to the instance. Additionally, the Clear() and Cardinality() methods allow programmers to clear the edge list and see how many edges there are in the list of the instance respectively.

The Node class also raises two events: CaptionChanged when its caption is changed, and TypeChanged when its type is changed. The first allows the control to know when to update the caption in the visual representation of the node. The second allows the control to refresh the visual representation of the node accordingly to the new node type.

4.1.2 The Edge Class

This class represents instances of edges of a given graph, in a conceptual form. Together with the `Node` class described in section 4.1.1, a coder can use the `Graph` class to create a conceptual representation of a graph. Figure 4.3 presents the class constitution.

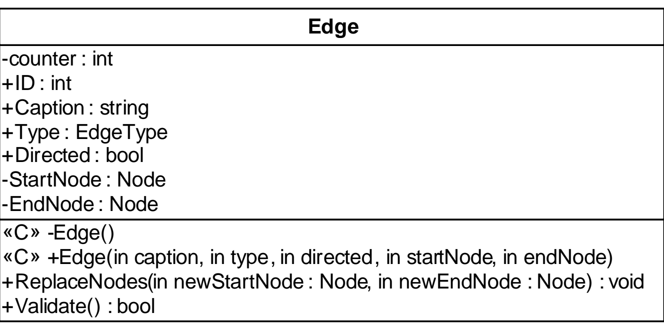


Figure 4.3: The Edge class.

The `ID` property acts as the object's unique identifier by incrementing the `static int counter` each time the class is instanced. `Caption` keeps information about the edge's label text caption, and is optional. The `Type` property defines what kind of relation this edge represents, and uses the `EdgeType` enumeration of the same class. This enumeration presents all the regular edge types existing in GISA: Hierarchical, Equivalence, Association and Temporal. The `Directed` property flags if the current edge has a defined direction from node `Start` to node `End`, or if the edge is simply undirected.

The class provides a private and a public constructor. The private constructor `Edge()` is test-oriented and internal-only, while `Edge(caption, type, directed, startNode, endNode)` allows external users to create a new, already filled edge.

Two methods were implemented: `ReplaceNodes(newStartNode,newEndNode)` replaces the current start and end nodes associated with the edge. `Validate()` checks if the edge has a starting and ending node. If it doesn't (if the start or end nodes are `null`), the edge isn't valid. This validation is used by the `Graph` class to validate a graph as a whole.

The `Edge` class also raises three different events: `DirectedChanged` when the `Directed` flag is changed, `CaptionChanged` when the caption is changed, and `TypeChanged` when the type is changed. The first two allow the control to know when to update the corresponding flags in the visual representation of the edge. The third is to allow refreshing of the visual representation accordingly to the new type.

4.1.3 The Graph Class

The `Graph` class represents a graph in its entirety, using both the `Node` and the `Edge` classes to define entities. The implementation of this class takes into account the needs of both the control and external applications when building graphs. Since it is the basic class used to define a graph digitally, it provides users and the control with tools to manage and control the information, rather that just tools to add and create a graph. These tasks include:

- Node and Edge addition: to add new instances to the graph;
- Node and Edge removal: to remove existing nodes or edges from the graph. If a node is removed, so are all the edges that are related to it;
- Node and Edge replacing: to replace existing instances with new ones;
- Validation: Validating a graph requires that no edges be present without assigned start and end nodes, that those nodes are all present in the graph, and that the graph has a defined start node;
- Acyclic check: to check if a graph has any cycles, i.e if there are any loops between adjacent nodes.

Figure 4.4 describes this class.

The `Name` property sets the name for the graph, and is optional. The `StartNode` refers to what node should be used as a navigation start point when the graph is loaded into the GIC. `CountNodes` and `CountEdges` simply return the number of nodes and edges in the graph. The private variables `nodes` and `edges` are `ArrayLists` of the `Node` and `Edge` objects that compose the graph.

Three constructors exist for this class: `Graph()` creates a new, empty graph with no name or start node. `Graph(name)` creates an empty graph with a specific name, and `Graph(name,startNode)` also adds a node to the node list and sets it as the start node.

`AddNode` and `AddEdge` are self-explanatory methods that add new objects to the graph. Nodes and edges can also be removed or replaced either by indicating the object itself or its unique `ID` using `RemoveNode`, `RemoveEdge`, `ReplaceNode` and `ReplaceEdge`. `GetNodeByIndex` and `GetEdgeByIndex` use the absolute object index in the graph list to get the correspondent object. `GetNode` and `GetEdge` get a node or edge based on its `ID`. `GetEdge`, however, also gives the option to find an edge given its start and end nodes. `Validate()` checks if the graph is valid by checking all its edges validity and the presence of a start node. `IsAcyclic()` checks if the graph is acyclic using the private method `GetLeaf()`. `Clear()` clears the whole graph

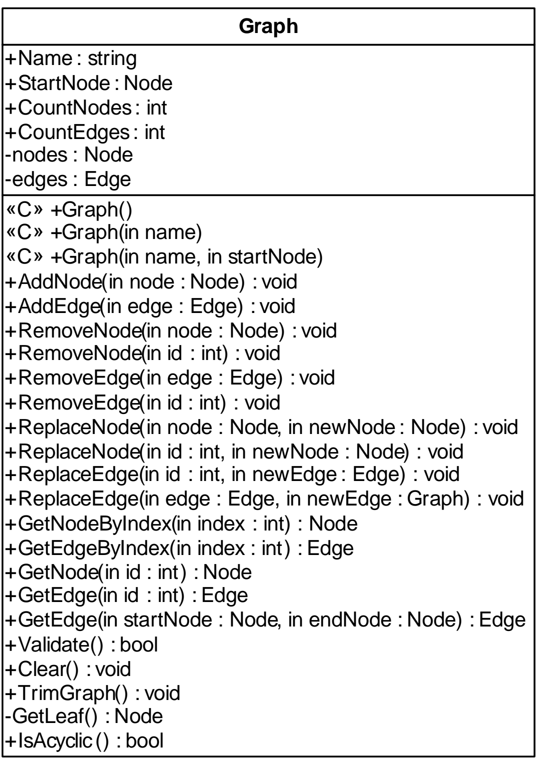


Figure 4.4: The Graph class.

content. `TrimGraph()` removes any unused space from the array lists containing nodes and edges.

A large number of methods were implemented to allow easy control and customization of a graph object by the programmers. When possible, several ways to perform one task were implemented: for example, one may replace a specific node by either indicating which node should be replaced, or simply by using the corresponding `ID`. Security measures were also implemented in this class to prevent errors and minimize the effect of erroneous elements. The list of nodes and edges of a graph, for example, is protected and can't be replaced like any regular property. Most functions also perform value-checking on the parameters they receive, stopping any changes on the data that may corrupt it.

The Graph class was though out to be used as a fill-in class. A programmer using the Calliope library should simply declare and construct a variable of type `Graph`, and then use `AddNode`, `AddEdge` and any complimentary functions needed to specify the contents of a graph. These contents are then easily loadable into the GIC.

4.2 The Athena Library

The Athena library is used to create a visual representation of a graph. Its classes provide a way to keep information about screen objects and the way they look. The two basic elements are shapes and lines. A shape can be rectangular or ellipsoidal in form, and has several other aesthetic attributes that define the way it looks. It is used to represent a node of a graph visually, and also contains information about attached lines. A line is a connector between two shapes, with some aesthetic parameters that define the way it shows to users, as well as pointers to its start and end shapes.

Class-wise, this means that a `Shape` object can have any number of `Line` objects attached to it, but that a `Line` can only (and must only) have two related `Shape` instances, as shown in Figure 4.5.

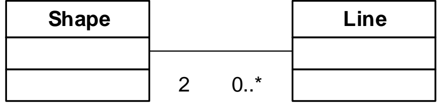


Figure 4.5: The Athena class diagram.

Typically, a shape will correspond to a node and a line will correspond to an edge in the GIC. The advantage of keeping both these representations separate is that each representation only needs to store information about details that are of importance to it. The color of a shape, for example, is of no importance to the node the shape represents. This guarantees modularity, so that users will have only the information they need for the classes they are using.

The next subsections depict the implementation for the `Shape` and `Line` class.

4.2.1 The Shape Class

The Shape class is used to represent nodes in a visual way. It keeps information about the location, size, and aesthetic look of concepts and entities. It basically describes to the GIC how one specific node should be presented to the user. At the same time, it also stores information about other entities that are connected to it through the graph, by the use of a simple list of Line objects. This way, when redrawing the scene after a node has been moved, the control knows what lines are attached to the shape and can redraw them too correspondingly. The class is presented in Figure 4.6.

The `ID` property identifies the object uniquely among shapes. It is calculated by incrementing the `static int counter` variable each time a new `Shape` is created. The `ArrayList Lines` contains the list of lines associated with a specific shape.

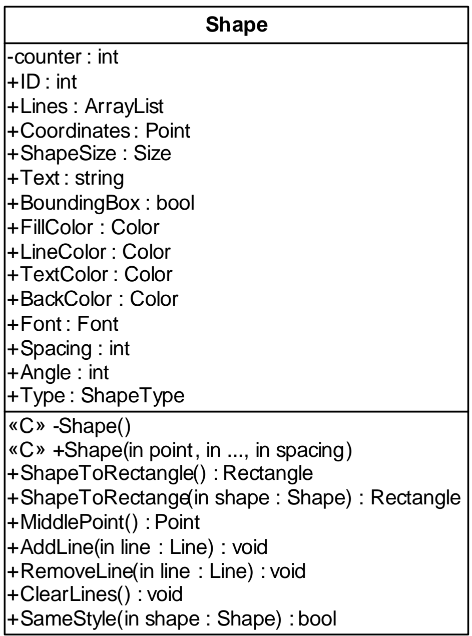


Figure 4.6: The Shape class.

`Coordinates` uses the System.Drawing `Point` class to specify the absolute position of the shape's top left corner in the control; `ShapeSize` uses the `Size` class to specify the width and height of the shape; the `Text` property contains the text for the shape's label. The `BoundingBox` flag is used to determine whether a bounding box should be drawn around the shape's text label; `FillColor`, `LineColor`, `TextColor` and `BackColor` keep information about the shape's fill color, line color, text color and bounding box back color; `Font` is the selected font for the text label; `Spacing` refers to the vertical distance that should be applied between the shape's bottom and the text label's top; `Angle` is the angle at which the gradient color (explained in section 4.3) will be applied for the shape; finally, `Type` uses the `ShapeType` enumeration to specify if the shape is either rectangular or a circular.

The class contains two constructors: a private `Shape()` for internal use, and a public `Shape(point, ... , spacing)` that fills all the aforementioned properties.

The `ShapeToRectangle` method returns a `Rectangle` object with info regarding the shape's bounding box, or the shape's bounding box related to another shape. The `MiddlePoint()` method returns a `Point` object with the coordinates of the absolute center of the shape. `SameStyle(shape)` compares the visual properties of the shape against the same properties of another shape, and checks if they match. All these methods are used by the control to determine what and where to draw the graph representation.

AddLine(line), RemoveLine(line) and ClearLines() are used to add, remove and clear lines from the Line ArrayList.

The Shape class also raises two events: TextChanged when the text is changed for the shape (so that the control can update the corresponding text in the information model); and a more generic event ObjectChanged when any other parameter is changed. The control uses this signal to raise its own event informing the program that a change was made in the visual representation of a graph.

4.2.2 The Line Class

The Line class implements the visual representation of a graph's edge, or connection between two distinct entities. It keeps information about the way this line is drawn in the control, as well as pointers to the two Shape objects it connects. This allows the control to know what nodes will need to be updated if a user wants to delete an existing line, for example. The class is presented in Figure 4.7.

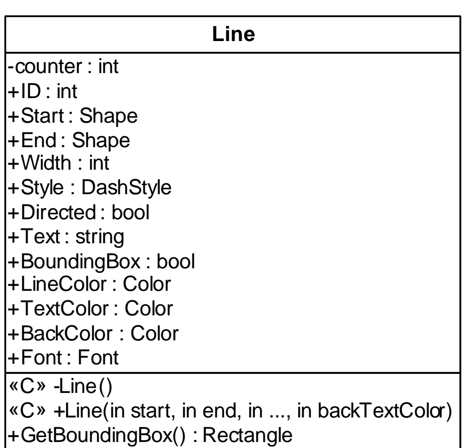


Figure 4.7: The Line class.

The ID property is used to identify an instance of the class uniquely, and is based on the incrementation of the static int counter each time a constructor is called; the Start and End properties point to the Shape objects that the line connects; Width is used to define the pixel width of the line when drawn; Style uses the System.Drawing DashStyle class to define the line's drawing style (solid, dashed, dotted, or other); Directed flags if an arrow cap should be drawn at the end of the line; Text is the string to apply at the line's label; the BoundingBox property specifies if a bounding box should be drawn around the line's label. LineColor, TextColor and BackColor set the colors for the line, the text label, and the bounding box's back color; the Font is used to define the font used in the text label.

The `Line` class provides a private and a public constructor: the `Line()` method is used only internally, while the generic `Line(start,end, ... ,backTextColor)` creates a new instance that fills the aforementioned properties with the passed parameters.

The class also implements an auxiliary method `GetBoundingBox()` that returns a `Rectangle` object with the bounding box that includes the line and the two shapes it connects, and is used by the control to define what areas to paint when shapes are moved around.

Finally, three events are raised by this class: `DirectedChanged`, `TextChanged` and `ObjectChanged`. They occur when the `Directed`, the `Text`, or any other property are changed. They are used to signal the control about these changed, so that changes in the data model can be made as needed.

4.3 The Kratos Library

Kratos is the third and most important library of the GIC, implementing the controller of the MVC paradigm presented in section 3.3.5, and using both the view and model implementations present in the Athena and Calliope libraries to manage, represent and interact with the user with graph information. Figure 4.8 shows the class diagram and the relationships between each of the interacting classes.

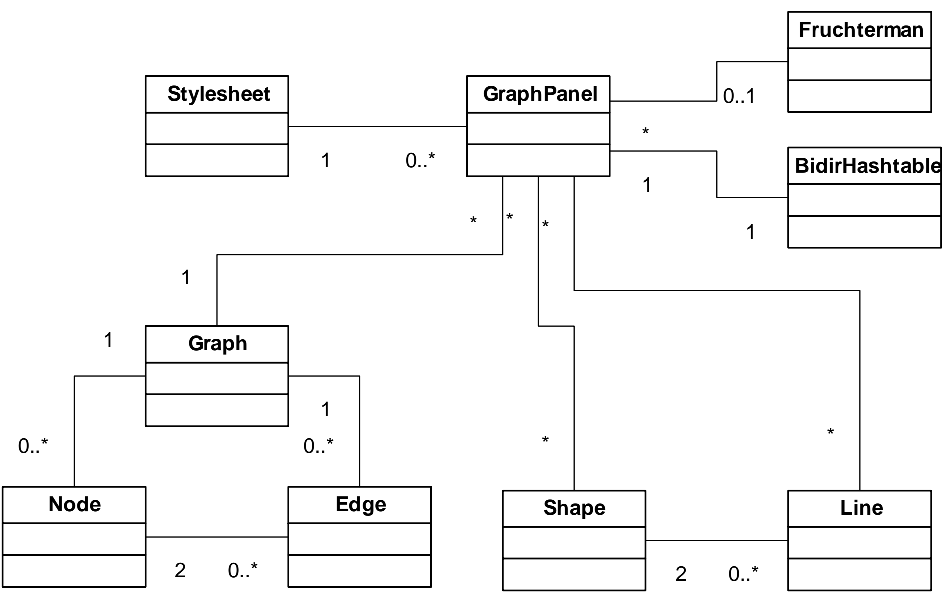


Figure 4.8: The Kratos class diagram.

The central class of the Kratos library is the `GraphPanel`. It is this class that represents the GIC and controls all the information flow between users, model and

view. This class keep information about one Graph instance at all times. It also keeps a list of Shape and Line objects that represent visually the objects present in the Graph object. Correspondence between Shape and Line objects and Node and Edge objects is kept at all times through a BidirHashtable, an open-source class that implements a bidirectional hashtable.

The GraphPanel class also points to one Stylesheet class that contains all the aesthetic information associated with each of the types defined for Node and Edge objects, as well as visual details that pertain the whole GraphPanel class. Finally, the Fruchterman class is an accessory class used to calculate the Fruchterman-Reingold algorithm.

The implementation of the Fruchterman and the BidirHashtable classes are not fundamental for understanding the way the GIC works, so a brief explanation of their purposes is given here. The first simply implements the algorithm described in section 2.5.3 for a given set of nodes. The second works just like a regular hashtable, except that it allows recovery of associated values in both ways. This means that both objects used in an entry of the table are considered keys. The BidirHashtable object used by the main class stores pairings *(Node,Shape)* or *(Edge,Line)* to allow quick access between the view and the data model.

The Stylesheet and the GraphPanel, however, are major pieces of the GIC. The next two subsections present these classes in detail.

4.3.1 The Stylesheet Class

The Stylesheet class is used by the GIC to store information about two things: the visual properties that should be used with the different types of nodes and edges, and the general properties that are considered general and should apply to all graphical elements, despite of their type. Figure 4.9 shows the structure of the class.

The class not only keeps information about how default and selected objects should appear to the user, but also provides the means to add new styles for custom types of shapes or lines. A new style is basically an instance of the Shape or Edge classes, whose properties can be changed to achieve the desired visual aspect. This is then recorded by the Stylesheet class via a string key that is used to match a type with a style. When the control is drawing an entity, it checks its type, and looks in the style sheet for styles whose key matches the entity type. If a style is found, the control applies it; if its not, the control uses the default style for the entity being drawn.

The DefaultShape and DefaultLine properties define the default visual details for shapes and lines; SelectedShape and SelectedLine define the same visuals for selected shapes and lines. The private properties shapeStyle and lineStyle both

Stylesheet
+DefaultShape : Shape +DefaultLine : Line +SelectedShape : Shape +SelectedLine : Line +BackColor : Color +GradientColor : Color +MinDistanceToLine : int +MaxTextLenght : int +ShapeType : ShapeType -shapeStyle : Hashtable -lineStyle : Hashtable
«C» +Stylesheet() «C» -Stylesheet(in defaultShape, in ..., in defaultShapeType) +AddShapeStyle(in type : string, in newStyle : Shape) : bool +AddLineStyle(in type : string, in newStyle : Line) : bool +RemoveShapeStyle(in type : string) : bool +RemoveLineStyle(in type : string) : bool +ReplaceShapeStyle(in type : string, in newStyle : Shape) : bool +ReplaceLineStyle(in type : string, in newStyle : Line) : bool +GetStyleForShape(in type : string) : Shape +GetStyleForLine(in type : uint) : Line

Figure 4.9: The Stylesheet class.

use a regular `Hashtable` to match records of new styles created with a string key. These provide support for the new types of styles that users may define.

The `BackColor` property defines the back color of the control; `GradientColor` is used to define the gradient start color that is used to paint the shape's fill; `MinDistanceToLine` is the minimum distance in pixels that the user's mouse must be from a line to show the line's label; the `MaxTextLength` property sets the maximum number of characters the text label for shapes and lines may have – if a text string is longer than that, it will be trimmed down; `ShapeType` defines the default shape for new shapes – rectangular or circular. These define the general properties of the visual appearance of graphs in the GIC.

Two public constructors are used: `Stylesheet()` creates a new, default style sheet; `Stylesheet(defaultShape, ... ,defaultShapeType)` creates a new style sheet with the properties set as those passed by parameter.

Some method were implemented so that the control or the programmer could manage information in a `Stylesheet` object. `AddShapeStyle(type,newStyle)` and `AddLineStyle(type,newStyle)` provide methods to add new shape and line styles. Similar methods exist to remove or replace a specific style. There are also two methods to recover a specific style for a given key.

4.3.2 The GraphPanel Class

The GraphPanel class is the main piece in the GIC. It serves both as the control implementation and as the manager for the graph information contained in the

model. It also builds and keeps the visual representation of the graph. The class details can be found in Figure 4.10.

Drawing of the visual representation of the graph is done through functions of the System.Drawing library chosen in section 3.3.3. These drawings are made using double buffering[2] to reduce control flickering, and using anti-aliasing to give shapes and lines a smoother feel.

The number of internal private methods used by the class is quite large. The inner workings of said methods is not necessary to understand how the control works. Instead, the following will explain how the control works and present all the public tools that the control presents to users, as well as the properties deemed necessary for that explanation.

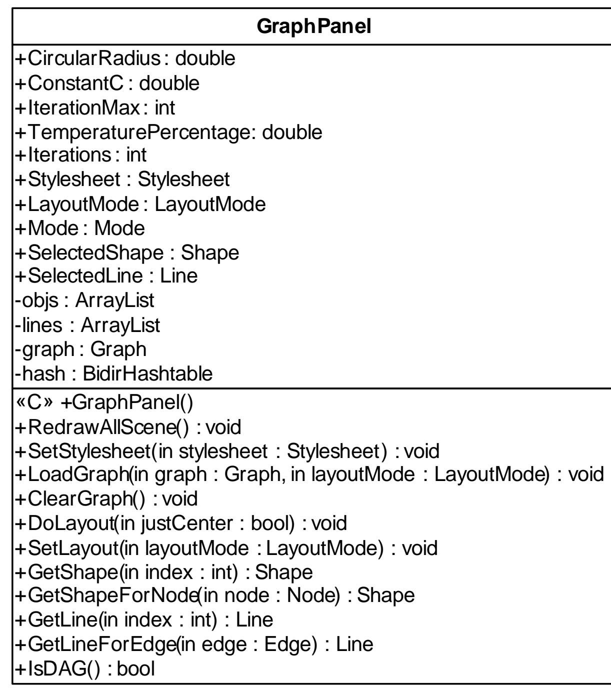


Figure 4.10: The GraphPanel class.

The `Stylesheet` property is used by the control to store information about how visual objects should be drawn. In each redraw cycle, the control takes each of the objects it is supposed to draw, checks its type, and looks for a corresponding style in the `Stylesheet`. If none is found, the default is used. If the user has manually changed a preset style, the control will keep the changes.

The `Mode` property uses the values from the enumeration with the same name. This mode sets the way the control should respond to users, using the interaction paradigm defined in section 3.3.4.2. The possible modes are:

[2] Graphical technique that uses a memory back buffer to reduce overhead on the actual picture redrawing

- **Edit**: edit mode means that the graph nodes are scattered around the control and that the user may move, edit, create or erase them. Concurrently, edges may also be created, edited or erased between nodes. Layouts may be applied at will, and no node is presented visually as selected (though one is selected); Nodes and edges are drawn accordingly to their types and to the styles defined in the `Stylesheet` property;

- **Navigate**: navigate mode means that the selected node will be drawn with the selected node style from the `Stylesheet` property. The graph is centered around the selected node in this mode, and if a new node is selected, the graph will center itself around it. Editing or moving the nodes is not allowed in any way, but applying layouts is;

- **Show**: show mode is similar to the navigate mode, except the graph is just centered in the control as a whole and no interaction is possible with it. Applying layouts is still possible.

The `LayoutMode` property sets which layout technique is to be used on the current graph. It uses the enumeration with the same name and that contains the layout techniques that were implemented for the GIC:

- **Fruchterman-Reingold layout**: this was the algorithm chosen in section 3.3.2 for representing thesauri graphs. Simple, fast and aesthetically pleasant, the result of the layout is calculated by the accessory class `Fruchterman`;

- **Radial layout**: this layout simply places the nodes evenly in a imaginary circle centered by the center of the control. It is a good way to find subjacent patterns present in the relations of a graph. Its implementation is simple, and is done by a private function inside the class. If the Mode property is set to anything other than the edit mode, the selected node is presented at the center of the circle, and not at the border of it;

- **Random layout**: this layout simply randomizes the location of the nodes of a graph, and is done by the `GraphPanel` class;

- **Orthogonal layout**: the original plan stated in section 3.3.2 was to implement a modified version of the Kamada-Kawai algorithm to achieve a pleasant layout for organograms. However, Kamada-Kawai doesn't allow for multiple roots to exist in a graph, so the algorithm was not implemented. Instead, a part of the Sugiyama algorithm was tentatively implemented. The algorithm only works with directed acyclic graphs, but allows for more than one root to be present. If the graph is directed and acyclic, the algorithm uses a simple algorithm to

layer the nodes according to their longest path to any root, and then distributes them evenly on each level. This corresponds roughly to the first two phases of the Sugiyama algorithm, and works for most of the basic cases of organograms. However, it is not complete, and the result is not always the most appropriate one. The algorithm also does not draw lines as orthogonal sets (more about this in section 5.3).

The way layouts work in the control is simple: when a redraw is forced, the `GraphPanel` class calls the appropriate method and receives the new positions of each node. It then uses the `Iterations` property to determine how many iterations should the animation between the current and the new positions have. The animation simply moves each piece a percentage of the displacement they must incur in a logarithmic fashion: the nodes appear to move faster at first and then slow down until a complete stop at their final coordinates. Appendix C presents how the same graph might look using each of the aforementioned layouts.

Information about the currently selected shape and line is kept by the `SelectedShape` and `SelectedLine` properties. This information is used to override style properties when the selected object is drawn and to draw it with the selected style. Although the `SelectedLine` property is implemented, it is not used.

`CircularRaidus`, `ConstantC`, `IterationMax` and `TemperaturePercentage` are properties that apply solely to the Fruchterman-Reingold algorithm and are passed by the control to the `Fruchterman` class when that layout is necessary.

A list of shapes and lines is kept by the control in the private properties `objs` and `lines`.These are instances of the `ArrayList class`. The actual graph is kept in the `graph` property, also private to prevent users from inadvertently changing something. Finally, `hash` is an instance of the open source class `BidirHashtable` (section 4.3) and is used to keep a match between view objects and model objects.

The only constructor for the class is `GraphPanel()`. The constructor starts the double buffering with the size of the control, and initializes any necessary variables.

The `RedrawAllScene()` method forces a redraw of the complete scene in the control. `SetStylesheet(stylesheet)` is used to load a new or modified stylesheet into the control. `LoadGraph(graph,layoutMode)` is the method used to load a `Graph` object into the control: not only does it become the actual graph, as the layout mode will be updated to what is specified in `layoutMode`. A complete redraw of the scene is also forced when a new graph is loaded. `ClearGraph()` clears the current graph in the control and allows a fresh start.

For applying new layouts or forcing the recalculation of the current layout, the control provides the `DoLayout(justCenter)` and the `SetLayout(layoutMode)`

methods. The first applies the current layout, using the `justCenter` flag to determine if the result should also be centered as a whole on the control; the second sets the new layout mode to be that expressed in `layoutMode`.

Some auxiliary methods have also been included to allow easy access to the objects being manipulated by any programmer. `GetShape(index)` and `GetLine(index)` can be used to get pointers to a shape or line in the `objs` or `edges` array lists. `GetShapeForNode(node)` and `GetLineForEdge(edge)` can be used to obtain a pointer to the shape or line that corresponds to a specific node or edge. Finally, `IsDAG()` can be used to check if the graph loaded on the control is a directed acyclic graph. This method is also used when calculating the orthogonal layout for a graph.

The `GraphPanel` class also raises a set of events meant to signal the programmer that important changes have been made with the information in the control. These events can then be used accordingly to the needs of the application using the control:

- `NodeSelected` occurs when a node/shape is selected;
- `NodeAltered` occurs when a node is changed in any way;
- `NodeCreated` occurs when a new node/shape is created;
- `NodeErased` occurs when a node is erased;
- `EdgeSelected` occurs when an edge/line is selected;
- `EdgeAltered` occurs when an edge is changed in any way;
- `EdgeCreated` occurs when a new edge/lin is created;
- `EdgeErased` occurs when an edge is erased;

4.4 The GIC Prototype

In addition to the GIC that was described in the previous sections, a small application was built using the .NET Framework 2.0 and the C# language to serve as a prototype wrapper of the control, and to showcase its capabilities. The implementation and overview of that application is now given.

The prototype was named *Olympus Graph Editor* in conformity with the project's codename. It is a simple program that allows the creation, edition, navigation and layout application of graphs. Users may load an example graph to the control or build one from scratch. The individual properties (both visual and internal) of graph entities can be changed with little effort, and the program also introduces a style sheet editor to create new styles or modify existing ones.

The program contains two GICs. One in the main program window, and another in the style sheet editor, used to preview the current style sheet. There are three classes:

- `MainForm`: the window form that contains an instance of the `GraphPanel` class and the tools to manage it;
- `Editor`: the class that implements the style sheet editor. It also uses a `GraphPanel` to preview styles;
- `Style`: a small class that is used by the `Editor` class to create new styles.

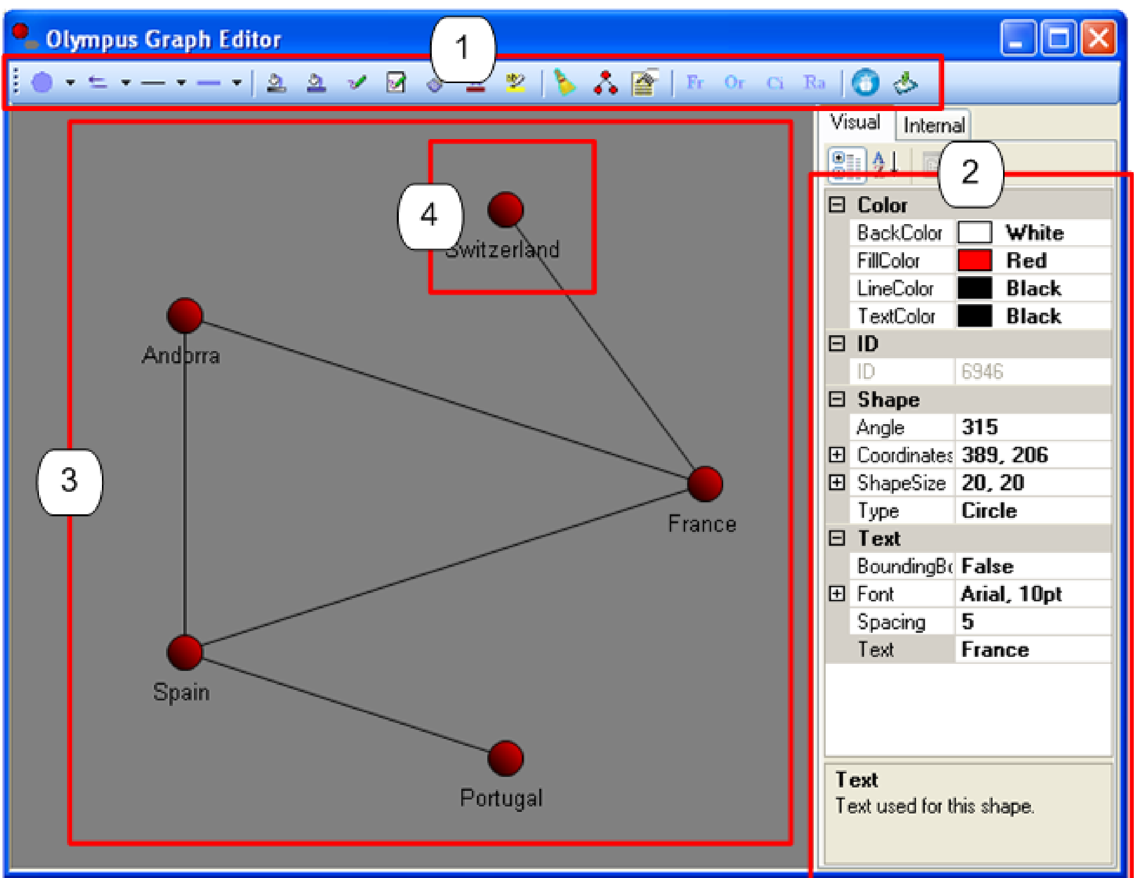


Figure 4.11: A screenshot of the GIC prototype.

Figure 4.11 presents an example screenshot of the application with a graph depicting the borderline connections of some European countries: Label 1 is the control bar used to change parameters, access the style editor, try the preset graph, apply layouts and choose the GIC mode to use; label 2 is the property editor, used to change both the visual and the internal properties of a selected graph object; label

3 shows the GIC itself, and the drawing area it encompasses; finally, label 4 gives an example of how a node might look like in the GIC.

The application starts in the edit mode by default, allows users to construct new graphs, but the mode, as many other parameters, can be changed easily. The control bar provides the following immediate tools to the user:

- Shape and Line formatting:
  - Default Shape: drop-down list allows selection of either square or circle as default shape;
  - Default Direction: drop-down list allows choice between directed and undirected new lines;
  - Default Line Width: drop-down list allows choice from 1px to 6px as default width for new lines;
  - Default Line Style: drop-down list allows choice from a selection of line styles as default for new lines.
- Color formatting:
  - Gradient Color: choses the color for the gradient of shapes;
  - Fill Color: chooses the color for filling the shapes;
  - Line Color: chooses color for new lines;
  - Shape Color: chooses color for the shape outer line of new shapes;
  - Control Back Color: chooses the background color of the GIC;
  - Text Color: chooses the color of text in text labels;
  - Text Back Color: chooses the color for the bounding box that may be shown behind text labels.
- Graph Options:
  - Clear Graph: completely erases any graph objects in the screen;
  - Load Preset Graph: loads a preset graph with four nodes to the control, cleaning any graph objects present;
  - Style Sheet Editor: opens the style sheet editor.
- Layout Options:
  - Fruchterman-Reingold: applies the FR algorithm to the current graph;

– Orthogonal Layout: applies the orthogonal to the current graph, if the graph is directed and acyclic (DAG). If its not, an error message is displayed;

– Circular Layout: applies the circular layout to the current graph;

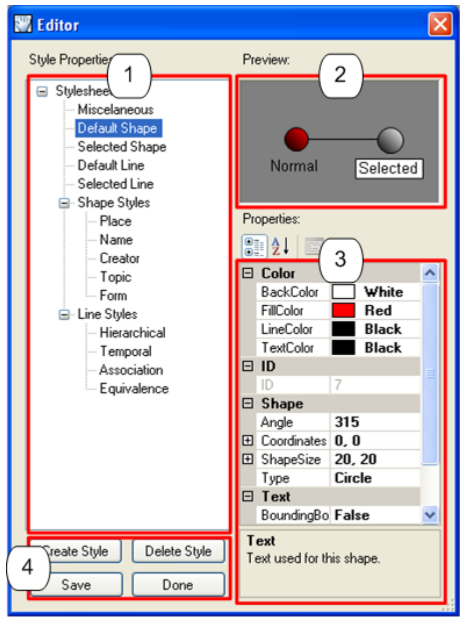
– Random Layout: applies a random placement for the current graph.

- Mode Options:

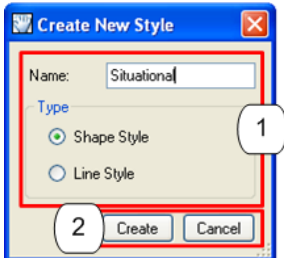
– Navigation Mode: switches the GIC to navigation mode;

– Edit Mode: switches the GIC to edition mode.

Interaction with the GIC is as described in section 3.3.4.2. Selecting a node or an edge by left clicking them will pass the node's information to the property editor. This editor has the *Visual* and *Internal* tabs on top: these switch between visual (shape/line related) and internal (node/edge related) properties for the selected object. Any change that directly influences the aspect of the selected object will be reflected on the control immediately. Concurrently, modifying properties in one model will update the other model if needed as well.



(a) The Editor Window.



(b) Creating a Style.

Figure 4.12: A screenshot of the style sheet editor.

The style sheet editor can be used to alter currently defined styles and to create new ones. Figure 4.12a shows a screenshot of the style sheet editor and its main control zones. Label 1 represents the style explorer. It presents a list of GIC's miscellaneous properties, preset styles, and custom made styles. Left clicking one

of the styles here makes it the selected style to edit. The first five items in the list represent generic GIC properties, default shape and line styles, and selected shape and line styles. The two last items can be expanded and contain custom made shape and line styles.

Label 2 shows the preview area. This zone contains a GIC that is used to show how the currently selected style appears. This GIC is set to view mode, which means no interaction is possible with it. It simply shows two test shapes (a normal and a selected one) and a line connecting them. Users can move the mouse near the line to see how the line style applies to the text label.

Label 3 is the property editor, similar to the one in the main window form. Once a style is selected in the style explorer, its properties will be shown here. Changes in this area will be reflected in the preview GIC.

Label 4 shows the control buttons area. The four available button allow users to create, delete and save styles, and to close the window and accept changes. The save button was not implemented in this version of the prototype, but it will eventually save style information to a file. The delete button will delete the currently selected style, as long as it is a custom one. A check message is presented before actually deleting the style.

Finally, the create style button will open a small window as shown in Figure 4.12b. This window allows users to define a new style by giving it a name and defining if the new style applies to shapes or lines, as shown in label 1. After this information is inserted, the user may create the style or cancel the creation process as shown in label 2.

Custom styles will then appear in the style editor's explorer. They will only be used if the name given to the style matches the type of object it was designed for. So if a new shape style is created with the name *Situation*, its aspect will only be used on shapes whose type is also *Situation*.

4.5 Final Remarks

With the project implementation explained, and the prototype of the GIC presented, some final remarks about the GIC and its inner workings are taken.

First and foremost, it is important to understand that the implemented version of the GIC is indeed only a prototype, far from what the final version should look like. Section 5.2 provides some insight into what objectives were and were not achieved during the course of the project. However, during the course of the twenty weeks of work, it has become clear that even after all the objectives set initially are achieved, this project will still have many possibilities of being developed further. Some reflexions on this matter are made on section 5.3.

Secondly, it was stated as an important objective since the very beginning that the GIC would be platform independent and work at least under Windows and Linux OS. Important choices during the course of the project were made in function of this requisite – namely the choice of the graphic library to use. In the end, the objective was achieved: the GIC can be run on Linux as well as in Windows if the Mono framework [38] is used to replace the System.Drawing methods utilized by the control.

Finally, the GIC has two simple requirements to be run: the .NET Framework 2.0 must be installed in the computer where the prototype application is to run; and that the Mono Framework is also installed if the application is to run on an operative system different from Windows.

Chapter 5

Conclusions and Future Work

In the previous chapters this report has focused on making a literature review, describing a methodology revolving around the use of the MVC architectural paradigm, and explaining the implementation of a graph oriented control meant to allow simple representation and interaction with graphs. This chapter summarizes on what was achieved during the course of this work and what are the future plans for the results of it.

Section 5.1 summarizes the work done during the project. Section 5.2 overviews a list of achieved objectives and failed goals. Section 5.3 makes some considerations about the future work for the project.

5.1 Work Developed

Through the course of twenty weeks, a reliable and easy to use graph interaction control was developed from scratch. A thorough literature review was made on several aspects regarding graph theory, interaction, layout algorithms, existing applications and drawing conventions. An in-depth list of objectives was then set, and a list of use cases was defined.

A methodology was then developed for the control using the model-view-controller architectural paradigm. The paradigm uses a data model to represent graphs, a view model to visually show the graphs, and a controller that unifies the two data models. This gives the solution reusability, maintainability and modular design. The control was created using the .NET Framework 2.0 and the C# language. The System.Drawing [37] graphic library from Microsoft was used for graphical plotting due to its low-level abstraction and ease of portability to other OS through the Mono Framework [38]. The data model chosen to represent graphs was the Incidence List.

Four layout techniques were implemented for the control, one of which is a direct implementation of the Fruchterman-Reingold [20] algorithm, and another which is a partial implementation of the Sugiyama-Misue [22] algorithm. A simple and effective interaction paradigm was also chosen for the control based on the developer's previous experience in the field.

All the decisions made regarding the methodology and tools to use have taken into consideration both the needs of ParadigmaXis and GISA and the time there was available to implement the control.

The implementation itself is composed of three separate libraries and a prototype application to showcase the control's functionality. The libraries represent the graph data model (Calliope), the visual representation model (Athena) and the controller (Kratos). Calliope uses the classes `Graph`, `Node` and `Edge` to represent graph information; nodes and edges have specific types associated to them. Athena uses the `Shape` and `Line` classes to represent graph objects on the screen. Kratos implements the `GraphPanel` class to manage information in the other two libraries and the `Stylesheet` class to keep information about styles applicable to visual objects according to their type in the data model. The prototype application wraps all the libraries and provides users with tools to edit, navigate and create new graphs using the control.

5.2 Achieved Objectives

The original list of objectives described in section 3.1 was successfully implemented only to a certain level. Some objectives became road map features as time passed, others were just dropped out due to the lack of available time. After the methodology was completely defined and the development of the control started, the objectives were all prioritized as very important, important, secondary and negligible. Unachieved objectives were mostly cause to the lack of time to develop, and to the fact that their priority was secondary or negligible.

The following list checks what objectives from the original list were and weren't achieved:

- Regarding basic features:
  - Graph loading, creation, editing and navigation were totally implemented as proposed in the original objective list. Graph loading is possible through code and is meant for programmers. Creation, editing and navigation of graphs are available to any user;
  - Four graph layouts were implemented. A layout for thesauri was successfully developed and two additional layouts were also implemented. A

specific layout for organograms was tentatively developed, but the results don't match those set by the original objective list;
  - Style sheets were implemented almost to a full extent. Original plans allowed saving and loading of already created style sheets. The prototype allows creation and use of style sheets, but does not allow saving and loading them;
  - A separate control for organograms displaying a temporal slide bar was not built, due to the lack of time and the fact that this was considered a low priority objective.

- Regarding the interaction paradigm:
  - The displaying of thesauri is standard compliant with the ISO 2778, following when possible the visual representations proposed in the document;
  - Measuring how user-friendly the interaction with the control is is rather subjective. The developer's previous experience with user interface was taken into account when designing the interface however;
  - Customization of the graph elements was developed to an extreme. Visual properties for both shapes and lines on the screen allows for a great diversity of aesthetic looks. One aspect was not developed though: the depth of nodes to present to the user when navigating through a graph. The prototype simply shows all the graph when navigating;
  - The control is entirely event-driven, as can be seen in the prototype application built.

- Regarding Graphical Libraries:
  - The chosen graphical library is free to use, and has a large thriving developer community. It provides tools to enhance performance (such as double buffering and anti-aliasing) and allows multi-platforming with the use of an additional framework;
  - Graphic acceleration was not implemented due to the lack of support for the chosen library.

- Regarding architectural pattern: the use of the MVC paradigm for architecture guarantees modular design, scalability and maintainability as originally intended;

- The C# language was used as originally intended;

- Multi-platforming was achieved with the help of the Mono Framework;

- Unit testing was not implemented due to the lack of time available.

Additionally to the achieved goals described in this list, there were some other points that were not mentioned in the original objective list and that ended up implemented in the prototype. Interaction with graph elements can be done using a context menu. This menu provides some tools such as the possibility to invert an edge's direction. Animation of the graph when applying layouts was also not in the original list of objectives, but ended up being implemented to improve user experience.

Style sheet loading and saving was intended to allow saving the style sheet information in a file using some sort of marking language such as XML. The objective was considered secondary as it was not vital to the correct working of the control.

The customization of the depth of nodes to show during navigation was not implemented due to lack of time. It is, however, a simple objective to achieve and one of the first on the current prioritized list of objectives.

The separate control for time oriented organograms was never even considered in the prototype version since the work with organograms is far from done. The biggest problem with the control in its current version is the inability to correctly lay most organograms due to the poor orthogonal layout implementation used. The fact that lines are not yet drawn as orthogonal in this version is also a major setback. Since the separate control should focus specifically on organograms, there was no interest in developing it for an incomplete representation of these structures.

Unit testing was since the beginning regarded as a secondary objective. It would add consistency and safety to the control, but since the work focused only on the development of a prototype, the implementation of unit testing became less important.

5.3 Future Work

As stated previously in the report, the GIC is far from complete. Many improvements and new features could still be implemented in a final and consistent version of the control. Together with the obvious debugging and testing the prototype still requires, the following list presents some of the objectives considered of most interest to the future development of the control:

- Loading and saving style sheets: using a markup language, the idea here is to allow saving style sheets and even evolving them into a rule-based language that could be uniquely applied to all the control;
- Organogram layout and orthogonal drawing: the development of the current orthogonal algorithm into the actual Sugiyama-Misue algorithm would allow

work with organograms to progress quickly. The next steps are orthogonal line drawing and the development of the control with the time slider bar for time-oriented organograms;

- Unit testing: unit testing is not present since the control is considered a prototype. A final version, however, would benefit greatly from the consistency unit testing brings; It is an essencial objective before integrating the GIC with GISA;

- Dynamic use of entity types: Style sheets allow the creation of styles for any number of types. The data model, however, only allows a preset set of types for nodes and edges. It would be interesting to evolve this model and add support to customizable lists of types;

- Dynamic loading of graphs: the prototype supports loading of new graphs into the control, erasing previous graphs. The next step is to allow dynamic loading of graphs that would keep nodes common to the current and the loaded graph, erase those that don't exist in the loaded graph and add only those that still don't exist in the control;

- New layout algorithms: it would also be interesting to see new layout algorithms to diversify even more the user experience with the control.

The possibilities for long-term development, however, are much bigger. Graphs interconnect several areas of research, and as time and new solutions appear, the functionality of the control could also grow much bigger. The following is a list of road map features that would be interesting to implement eventually:

- New shapes: hexagon, triangle, or polylinear shapes;

- New customization properties: new gradients, arrow types or color properties;

- Three dimensional graph interaction: the use of a three dimensional space to represent graphs, hypergraphs and more complex structures;

- Graphic acceleration: to provide increased performance for the control;

- Search algorithms: implementing algorithms that don't focus on how graphs look but rather on what information graphs contain;

- Generalization: use the control not only for graphs, but also for other, more general concepts such as drawing and networking.

On a final remark, the author would like to add that the addition of any of these plans would also depend greatly on the type of application using the control. Although shown as part of an application prototype, the control is the core of the project and the part of it that truly has the potential to evolve into whatever external applications may need.

References

[1] Paradigmaxis s.a. - site oficial, 2008. Online at http://www.paradigmaxis.pt/, last visited on 27/06/2008.

[2] Gisa - site oficial, 2008. Online at http://gisa.paradigmaxis.pt/, last visited on 27/06/2008.

[3] L. Euler. Commentarii academiae scientiarum imperialis petropolitanae, 1736.

[4] Reinhard Diestel. *Graph Theory: Electronic Edition 2005*. Springer, August 2005.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[6] Alexander K. Hartmann and Martin Weigt. *Phase Transitions in Combinatorial Optimization Problems*. John Wiley & Sons, february 2008.

[7] Bruno Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, 1998.

[8] Graph drawing symposia webpage, March 2008. Online at http://graphdrawing.org/, last visited on 12/03/2008.

[9] Isabel F. Cruz and Roberto Tamassia. Graph drawing tutorial. Technical report, Worcester Polytechnic Insitute and Brown University, unknown year.

[10] Peter Eades and Kozo Sugiyama. How to draw a directed graph. *J. Inf. Process.*, 13(4):424–437, 1990.

[11] Maria Eugénia Matos Fernandes e Rute Reimão Fernanda Ribeiro. *Universidade do Porto. Estudo Orgânico-Funcional: modelo de análise para fundamentar o conhecimento do Sistema de Informação Arquivo*. Reitoria da Universidade do Porto, June 2001.

[12] ISO 2788/NP 4036. Tesauros monolingues: Directivas para a sua construção e desenvolvimento, November 1992.

[13] Danny Heap. Graph representation. Notes for the algorithms and data structures course, University of Toronto, unknown year.

[14] Simon Dobson. Representing graphs in programs. Technical report, UCD Dublin, 2005.

[15] Kozo Sugiyama and Kazuo Misue. Visualization of structural information: Automatic drawing of compound digraphs, April 1991.

[16] Giuseppe D. Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, October 1994.

[17] Susan Sim. Automatic graph drawing algorithms. Technical report, University of Toronto, December 1996.

[18] Peter Eades. A heuristic for graph drawing. In *Congressus Numerantius vol. 41*, pages 149–160, 1984.

[19] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, 31(1):7–15, 1989.

[20] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, 1991.

[21] Eugene I Butikov. Regular keplerian motions in classical many-body systems. Technical report, St Petersburg University, May 2000.

[22] S. Tagawa K. Sugiyama and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. on Systems, Man, and CyberneticsSMC-11(2)*, pages 109–125, 1981.

[23] M. Siebenhaller M. Eiglsperger and M. Kaufmann. An efficient implementation of sugiyama's algorithm for layered graph drawing. *Graph Algorithms and Applications*, 9(3):305–235, 2005.

[24] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19(3):214–230, 1993.

[25] Oliver Bastert and Christian Matuszewski. Layered drawings of digraphs. In *Drawing graphs: methods and models*, pages 87–120, London, UK, 2001. Springer-Verlag.

[26] Wilhelm Barth, Michael Jünger, and Petra Mutzel. Simple and efficient bilayer cross counting. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, pages 130–141, London, UK, 2002. Springer-Verlag.

[27] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Trans. Graph.*, 15(4):301–331, 1996.

[28] Janez Brank. Drawing graphs using simulated annealing and gradient descent. Technical report, Department of Knowledge Technologies, Jozef Stefan Institute, undated.

[29] Human-Computer Interaction Lab. Piccolo home page, 2008. Online at http://www.cs.umd.edu/hcil/jazz/, last visited on 01/04/2008.

[30] Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. Toolkit design for interactive structured graphics. *IEEE Trans. Softw. Eng.*, 30(8):535–546, 2004.

[31] Orbiworks. Netron library, a diagramming library for .net 2.0, 2008. Online at http://www.orbifold.net/netron/, last visited on 01/04/2008.

[32] Orbiworks. Unfold, a wpf diagramming tool, 2008. Online at http://www.orbifold.net/Unfold/, last visited on 02/04/2008.

[33] Danyel Fisher Joshua O'Madadhain and Tom Nelson. Jung – java universal network/graph framework, 2008. Online at http://jung.sourceforge.net/, last visited on 01/04/2008.

[34] T. Nelson J. O'Madadhain, D. Fisher, S. White, and Yan-Biao Boey. The java universal network/graph framework (jung): A brief tour, 2008. Online at http://jung.sourceforge.net.

[35] Berkeley Institute of Design. prefuse — interactive information visualization toolkit, 2008. Online at http://prefuse.org//, last visited on 01/04/2008.

[36] Jeffrey Heer, Stuart K. Card, and James A. Landay. prefuse: a toolkit for interactive information visualization. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 421–430, New York, NY, USA, 2005. ACM.

[37] Microsoft Corporation. System.drawing namespace, 2008. Online at http://msdn2.microsoft.com/en-us/library/system.drawing.aspx, last visited on 01/04/2008.

[38] Mono official page, 2008. Online at http://www.mono-project.com/Main_Page, last visited on 03/07/2008.

[39] Cairo official website, 2008. Online at http://cairographics.org, last visited on 01/04/2008.

[40] Øyvind Kolås. Custom widgets using gtkcairo, 2004.

[41] Carl Worth. Cairo: Making graphics easy to print, 2005.

[42] Graphviz website, 2008. Online at http://www.graphviz.org/, last visited on 02/04/2008.

[43] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.

[44] Graham J. Wills. Nicheworks - interactive visualization of very large graphs. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 403–414, London, UK, 1997. Springer-Verlag.

[45] Janet M. Six and Ioannis G. Tollis. Circular drawings of biconnected graphs. In *ALENEX '99: Selected papers from the International Workshop on Algorithm Engineering and Experimentation*, pages 57–73, London, UK, 1999. Springer-Verlag.

[46] Alan Dix, Janet E. Finlay, Gregory D. Abowd, and Russell Beale. *Human-Computer Interaction (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[47] Juliele A. Jacko. *Human-Computer Interaction Handbook*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, 2002.

[48] Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, 1983.

[49] Dim Vision. The musicmap project, March 2008. Online at http://www.dimvision.com/musicmap/, last visited on 28/03/2008.

[50] Barret Lyon. The opte project, March 2008. Online at http://www.opte.org/history/, last visited on 28/03/2008.

[51] Websites as graphs - website, March 2008. Online at http://www.aharef.info/static/htmlgraph/, last visited on 28/03/2008.

[52] Touchgraph oficial website, March 2008. Online at http://www.touchgraph.com/, last visited on 28/03/2008.

[53] ThinkMap Co. The visual thesaurus, March 2008. Online at http://www.visualthesaurus.com/, last visited on 28/03/2008.

[54] We feel fine, March 2008. Online at http://www.wefeelfine.org/, last visited on 28/03/2008.

[55] Information Architects. Web trend map beta 2008, March 2008. Online at http://informationarchitects.jp/web-trend-map-2008-beta/, last visited on 28/03/2008.

[56] R. L. Mack J. K. Carroll and W. A. Kellogg. Interface metaphors and user interface design. In *Handbook of Human-Computer Interaction*, pages 67–85. Elsevier Science, 1988.

[57] Christopher D. Wickens, John D. Lee, Yili Liu, and Sallie Gordon-Becker. *Introduction to Human Factors Engineering (2nd Edition)*. Prentice Hall, 2 edition, November 2003.

[58] Stuart Hansen and Timothy V. Fossum. Refactoring model-view-controller. *J. Comput. Small Coll.*, 21(1):120–129, 2005.

[59] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

Appendix A

Types of Diagrams in ISO 2778

In section 2.3 the ISO 2778 [12] for the establishment and development of monolingual thesauri was used as source of information about how to best represent thesauri as graphs. The standard presents two possibilities for visual representation of thesauri: tree-like schemas and arrow-like schemas. An example of both is presented in this appendix.

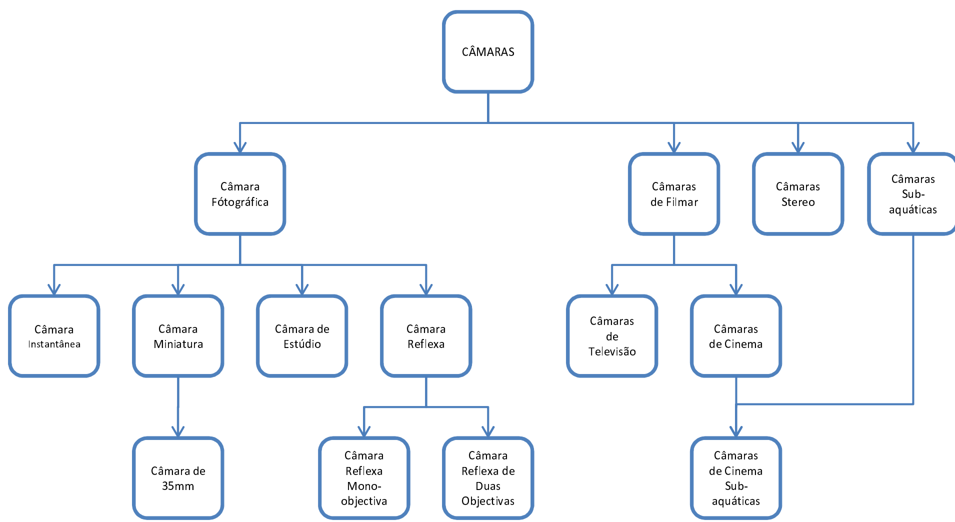


Figure A.1: Example of a tree-like schema.

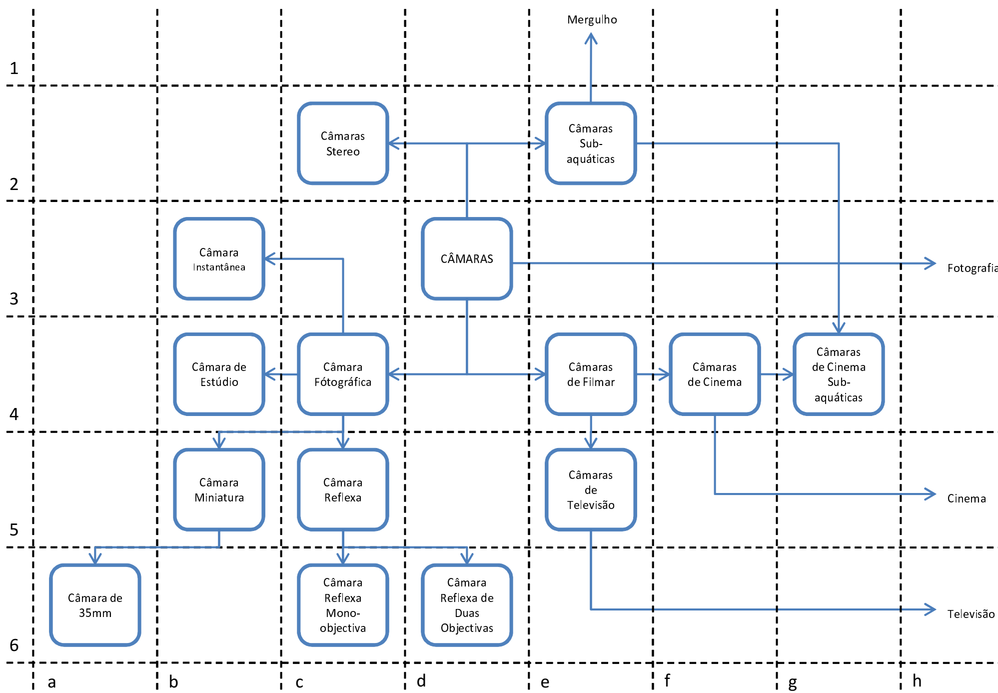


Figure A.2: Example of a arrow-like schema.

Appendix B

Class Diagrams

Class diagrams are used several times through chapter 4 to present and introduce the properties and methods available in a class. This appendix serves the purpose of presenting an example class diagram and the terminology to read said diagrams. Figure B.1 represents the diagram for the example class.

Example
-privateAttribute : int -anotherPrivateAtribute : string +PublicAtribute1 : object +PublicAtribute2 : bool
«C» +Constructor() «C» +ConstructorWithParameters(in param1, in param2) +ExampleMethod(in param1 : double, in param2 : bool) : int +MethodWithManyParameters(in p1, in ..., in pN)

Figure B.1: A class diagram example.

The class diagram is divided in three horizontal bars:

- **Class Name**: in this case, the class name is *Example*.
- **Property List**: the rectangle directly beneath the class name provides a list of properties for the class.
- **Method List**: the rectange directly beneathe the property list provides a list of method implemented in the class.

The following terminology is used to represent properties and methods:

- The plus and minus sign behind each property or method represent its visibility to other classes. A - means the property or method is private. A + sign means that the item is public;
- The $\ll C \gg$ tag behind a method means that method is a constructor for the class;
- Private properties start with lowercase letters. Public properties start with uppercase letters. Methods always start with uppercase letters.

- The parameter list of constructors does not state the datatype for each parameter, but the parameter list for methods does. Additionally, if the parameter list for any item is very long, the list will be abbreviated like the `MethodWithManyParameters` method shows.
- Parameters with the *in* tag before the name are input parameters. Those with the *out* tag are output parameters.
- The datatype for a property is set after the property name. The return datatype for a method is set before the method name.

Appendix C

Layout Algorithms in the GIC

This appendix presents how a test graph is laid out by the four layout methods implemented in the GIC. The four layout methods are: Fruchterman-Reingold layout, Radial layout, Random layout and Orthogonal layout. The graph used represents a set of European countries and the border adjacency between them. To allow the orthogonal layout to work with this graph, the adjacencies have been given direction in such a way that the graph is considered a DAG.

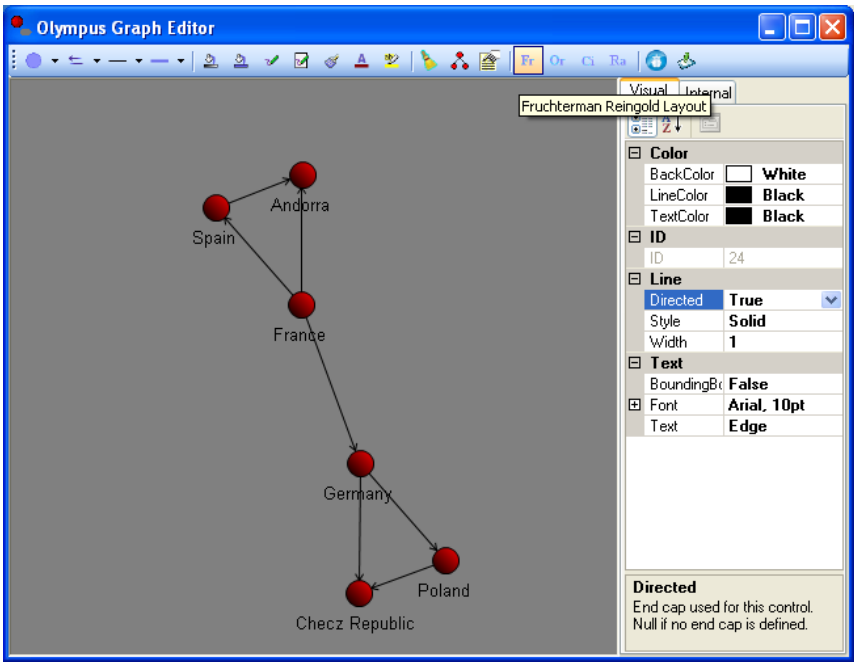


Figure C.1: Fruchterman-Reingold layout.

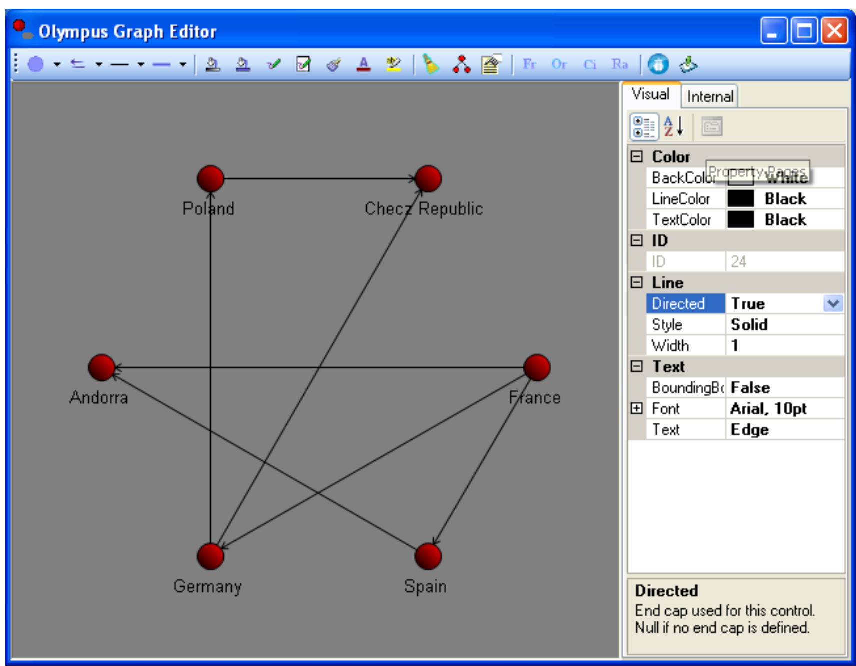


Figure C.2: Radial layout.

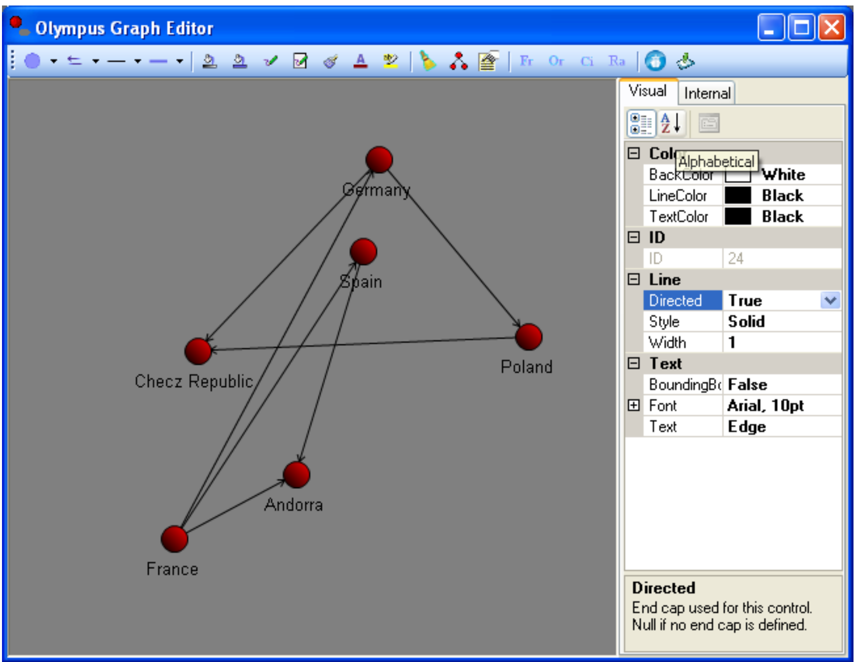


Figure C.3: Random layout.

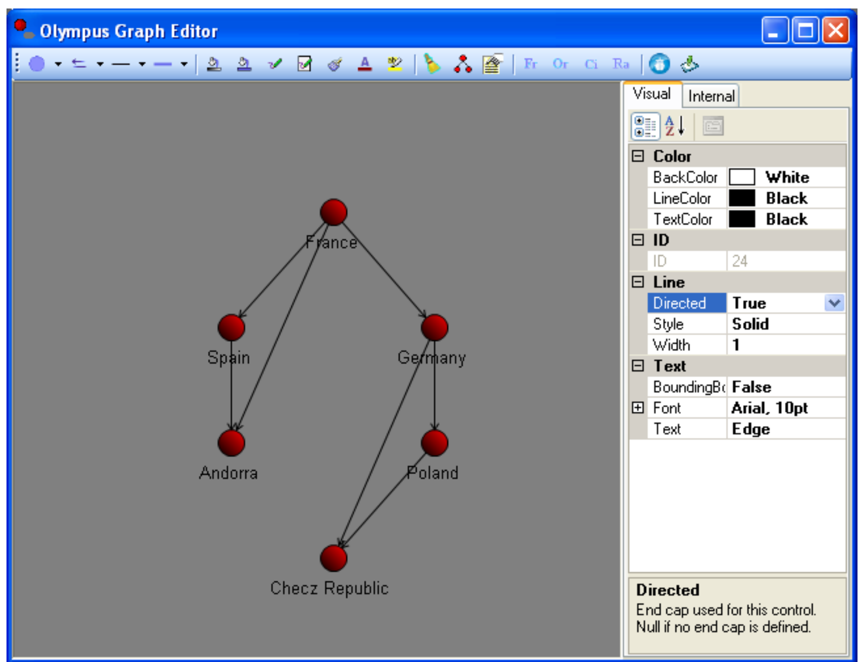


Figure C.4: Orthogonal layout.