FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

CleanGraph - Graph Viewing and Editing for Family Trees and UML Class Diagrams

António Alexandre de Almeida Martins

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Daniel Castro Silva

July 30, 2021

CleanGraph - Graph Viewing and Editing for Family Trees and UML Class Diagrams

António Alexandre de Almeida Martins

Mestrado Integrado em Engenharia Informática e Computação

July 30, 2021

Abstract

The importance of information cannot be understated in our society. It is required for decision making, be it in politics, economy or any other area where making an informed choice is imperative. As such, the availability and correct presentation of data are crucial.

With information growing in quantity and in complexity, it becomes harder to present it in its entirety while remaining intuitive. Information visualisation systems can represent this information by using graphs of varying forms and styles. Unfortunately, as the complexity of the data presented increases, so does the complexity of the diagrams, making interpretation more challenging. This issue is further exacerbated by the limited interaction afforded by many of the representation platforms, and these problems extend to different types of data and graphs. For this work Family Trees and Universal Modelling Language (UML) class diagrams were chosen, as both of these data types are frequently represented graphically and suffer from the interpretation issues alluded to previously.

Family trees and UML class diagrams applications commonly rely on static graphic representations of the data, limiting interaction to simple panning and zooming. Additionally, when displaying more significant amounts of data, some resort to hiding information, leading to the creation of misleading graphs and further misinterpretation of data.

The main focus of this work was addressing these issues by creating a platform capable of generating diagrams where all data is visible while avoiding overlapping, and having improved methods for manipulating information. These diagrams prioritise clarity by reducing visual clutter and by highlighting important information. Furthermore, it gives users new ways to handle data, such as selecting elements to receive more information, spotlighting relevant parts of a graph and allowing complete control of the representation.

To ensure the viability of the solution, three main tasks were devised: (1) defining an appropriate diagram disposition and why it is the more advantageous; (2) understanding the best way to create interactions with diagrams while making it intuitive and readable; (3) finding how to make the platform usable for the intended audience. The byproduct of these tasks was the creation of *CleanGraph*, a modular graph viewer and editor capable of handling both family trees and UML class diagrams.

With usability in mind, the platform was developed as a web application compatible with phones, tablets, and computers, designed with a server to offload more intensive tasks such as importing files and generating diagrams. This was done to minimise the hardware requirements of devices meant to use the platform and create a multi-platform experience allowing users to shift between their devices. Being compatible with more than just computers broadened the input capabilities, allowing for interactions with a cursor, keyboard and multi-touch touchscreens. The diagrams are generated with visual clarity and completeness as a priority with a graph layout algorithm designed to create an apt representation, minimising issues such as overlapping, hidden or duplicated information. By being modular at its core, the implementation of new graph types and features is facilitated, which gives a lot of potential for the platform.

Testing the effectiveness of *CleanGraph* was done via an evaluation form with a set of tasks to compare it against other platforms in a real-world scenario with individuals with an interest in genealogy. The UML class diagram module went untested as there weren't enough relevant features implemented that would require testing. From the tests results, it was possible to conclude that the platform implemented achieved the expected, with test subjects performing 40% better in the selected tasks and reducing the number of mistakes by 50% compared to other platforms.

In conclusion, *CleanGraph* can become a better tool for users seeking improved methods to view their ancestry data, as well as software architects, project managers and system analysts, who would benefit from a better platform for system representation. The developed features can help them convey their ideas easily to developers or anyone interested in the data represented, improving simplicity, workflow and potentially bringing new and improved methods of viewing and displaying information in these areas.

Keywords: Diagrams, Interaction, Unified Modelling Language Diagram, Family Trees, Graph Layout, Information Visualisation

Acknowledgements

I want to thank my supervisor, Professor Daniel Castro Silva, for guiding me through this dissertation, supporting me through planning and eventual realisation of the project and helping me with decisions with his experience.

I would also like to thank my parents, Ângelo and Telma, my sister Beatriz, my grandparents António and Vitória and my girlfriend Lara Dias, for their continued support throughout the years, and my friends, Antero Gandra, Diogo Teixeira, Fábio Azevedo, Felipe Lemos, Flávio Couto, Gonçalo Correia, Guilherme Amaro, João Esteves, José Diogo, Márcia Meira, Mariana Dias, Mariana Neto, Ricardo Lopes, Tiago Santos and Vitor Magalhães for helping me deal with all the work and keeping me both happy and encouraged to keep going. To all the mentioned people, I am very glad to have had you all by my side through this adventure. Thank you.

António Martins

"There can be miracles
When you believe."

The Prince of Egypt

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
CSV	Comma-Separated Values
CSS	Cascading Style Sheets
FT	Family Tree
GEDCOM	Genealogical Data Communication
HTML	HyperText Markup Language
InfoVis	Information Visualisation
JS	JavaScript
JSON	JavaScript Object Notation
JUNG	Java Universal Network Graph
OMG	Object Management Group
OMT	Object-Modeling Technique
OOSE	Object-Oriented Software Engineering
PC	Personal Computer
PDF	Portable Document Format
SUS	System Usability Scale
SVG	Scalable Vector Graphics
UI	User Interface
UML	Unified Modelling Language
VR	Virtual Reality
WIMP	Window Icon Menu Pointing device
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Chapter 1

Introduction

The need for representation of information has existed ever since humans wanted to share their knowledge about a subject or system. [Friendly and Wainer, 2021] One of the many ways to represent it is through graphs, and over the years different types of graphs were created to display information in an optimal and clear way for their intended audience. [Ware, 2012] Diagrams are the omnipresent representation type in all fields of research and their correct use allows for better and easier understanding of information [Chen et al., 2009] [Riaz and Ali, 2011].

The correct interpretation of data is a necessity for many individuals, businesses and data analysts and as such data must be as easy to explore as possible. On its own, raw data can be hard or even impossible to analyse; it is the displaying of the information in an appropriate way that makes its contents perceivable by the targeted user-base [Purchase, 2014].

The systems created to give users the ability to correctly analyse information are usually called Information Visualisation (InfoVis) Systems [Sorapure, 2019]. They are created with the goal of displaying data that could not be easily assessed and compared onto a platform that can facilitate its interpretation.

They can be divided into two separate but deeply interconnected parts, the representation of the data and the interactions with it.

The presentation of data can be done in many different ways, making the selection of the most apt method essential for making the interpretation as easy as possible. Different data types lend themselves to be better represented by certain types of representations [Kaidi, 2000].

Moreover, different representations of data can lead to different methods of interaction with it. All in all, interaction is an essential part, or even a requirement, for users to understand the data in a system [Kosara et al., 2003]. However, this factor is often overlooked. InfoVis systems try to find an optimal method for presenting information, but often do so in a static or non-intuitive way, leaving room for improvements. By giving users the agency to manipulate the visualisation, they are more capable of understanding the data they were shown [Yi et al., 2007].

1.1 Context and Motivation

InfoVis systems are used across many platforms to represent data, however, some of these systems have issues representing what they are supposed to display. The two types of graphs where some of these issues need to be tackled are Family Trees (FT) and Unified Modelling Language (UML) class diagrams. The main issues with commonly used platforms for these graph types include the overuse of traditional input and interaction methods, as well as reliance on static graph layouts or incomplete representations. Although this does not make them incapable of doing their designed purpose, it leaves a lot of potential for development.

This work focus on improving these aspects of graph visualisation and interaction by providing not only full graph representation, but also new tools for their exploration. The aim is to take a new step forward in this area, with the ultimate goal of improving user experience and satisfaction when working with FT and UML representation.

1.2 Objectives

To improve upon the existing types of InfoVis systems and provide an alternative that can give apt representation and interaction with data, a solution was developed specifically targeting two different data types: Family Trees (FT) and Unified Modelling Language (UML) class diagrams.

Ensuring that the created platform would provide appropriate results required following a set of defined goals [Victorelli et al., 2020] [Leeuw and Michailidis, 2000]:

- **Data Representation Accuracy** — The graphs created must follow the rules set by the data type it is portraying. If no rules for the format are specified, the representation should try and be as informative and clear as possible.

- **Interaction** — In addition to interactions such as panning and zooming, other navigation features such as tools to move parts of the representation or search for elements must be implemented to provide an improvement over other platforms.

- **Availability** — The solution must be developed to support the devices used by the targeted audience, as well as their input capabilities. Furthermore it must keep performance requirements to a minimum, enabling support for less powerful devices.

- **Ease of use** — This work seeks to make an improvement for information interpretation in graphs, and as such, the resulting solution must provide a user experience and graphical representation comparable or better than existing ones.

- **Compatibility** — As much of the data available is archived in different file types, the solution must allow importing it from at least one broadly supported file type for each of the graph styles.

Creating a new representation platform for the two graph types requires the implementation of several components for the system: a database to enable account creation and saving diagrams, a server to do most of the processing, an API to handle requests from the clients and a web-based client, from where most of the features of the platform are assessed.

Finally, the platform is tested in with the intended target audience, in order to evaluate its performance and ease of use in a real-life scenario.

1.3 Dissertation Structure

Along with the introduction, this document is comprised of five more chapters.

In Chapter 2 the related work is discussed, comprised of the different existing platforms, their implementations of the different types of graphs and their advantages and disadvantages. Also, input methods and technologies used for information systems currently available at the time of the writing of this document are compared.

Chapter 3 discusses common problems in the graphical representations and a decision is made towards the implementation of certain features for the platform developed, according to previous research.

Chapter 4 comprises implementation decisions taken when developing the platform, as well as how its functioning parts work together. Additionally, the platform features are specified in how they function and what issues they are meant to address.

The testing methodology and test results are analysed in chapter 5.

Finally, conclusions and future work are included in Chapter 6.

Chapter 2

Related Work

This chapter focuses on the current state of technologies in Family Tree and UML Class Diagrams visualisation, as well as current solutions proposed to solve common issues. This understanding is essential in the development of a new and improved solution.

2.1 Representations to be improved

The two types of representations chosen for the purpose of this work were Family Trees and Unified Modelling Language class diagrams, due to their shared similarities both in representation and in issues to tackle.

In graphical representations of family trees, as opposed to text based, there are several components needed to accurately portray the plethora of information available about a given family.

The representation is composed of people connected via relations. These relations can be of different types: marriage, divorce, non-marital partnership, biological child, adoptive child or conception of a child, and as such need to be represented differently. Additionally, some of these relations are not exclusively between people; for example, the birth or adoption of a child can be represented by the connection between a child and the relation between their parents.

Likewise, UML class diagrams are made of UML classes that are connected via relationships. These relationships have specific visual characteristics that help distinguish between the different types. There are also special types of classes, association classes, that form relationships between themselves and other existing relationships.

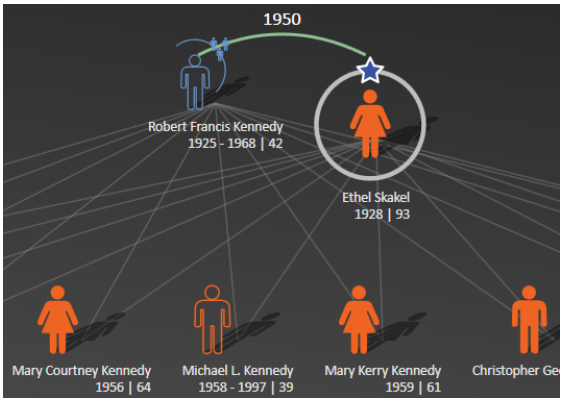
As such, due to the shared node-edge and edge-edge connections between these two graphical representations, visualisation and interaction can also be shared between both, making them suitable to use with the same representation framework.

2.2 Family Tree Representation

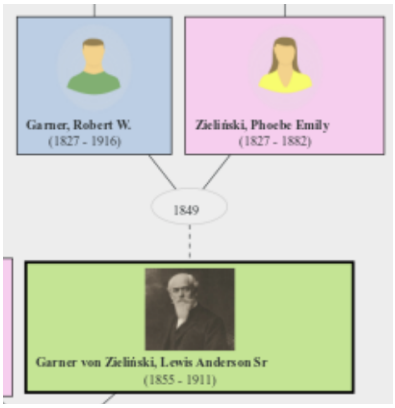
Family tree diagrams do not follow a standardised set of rules for diagram and information representation. Different platforms decide on how a family tree is created, how the elements are laid

out and what information is displayed. Although no standard exists, there are common elements across most family tree representations. Studied representations of family trees all contain two main elements: representations for people, and representation for relations or families. How these elements are shown in a graph vary greatly with some platforms choosing to offer more information through the use of labels, colour and images.

In family trees, families are traditionally represented by groups of people connected between each-other to form relations and connected to their descendants and ancestors. How the descendants and ancestors are connected to a person depends on the platform, some choosing to connect each parent to the child (Fig. 2.1a), some connecting parents to a common relationship node from which descendants are generated (Fig. 2.1b).

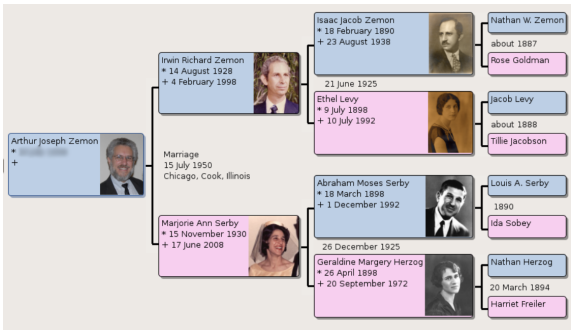


(a) Both connections to descendants in *FamilyShow*.

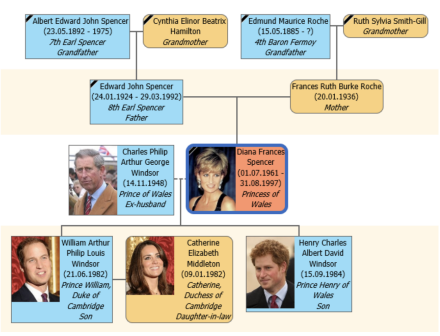


(b) Common node for descendants in *Gramps*.

The layout of these elements also varies greatly favouring different types of exploration and different overviews of the family. Pedigree Trees (Fig. 2.2a) are an example of this kind of behaviour, they favour exploring ancestors of a single element of a family tree, but provide little to no help when comparing people of the same generation, but it can be expanded to allow for more use cases [Santos et al., 2013]. There are also more complete trees like *Agelong* is able to produce (Fig. 2.2b) that are composed of people and relations, but allow for a more complete representation.



(a) Example of a pedigree tree in *Gramps*.



(b) Example of a more complete tree in *Agelong*.

2.3 Existing Family tree representation platforms

Several different genealogy platforms are available and will be described in the following sections. These platforms vary in how the information is presented, how the graphs are created and what features and interactions types are implemented.

2.3.1 Gramps

One of the programs used by people in the Genealogy community is *Gramps* (Fig. 2.3), an open-source program with a modular design and support for add-ons. [Allingham, 2020].

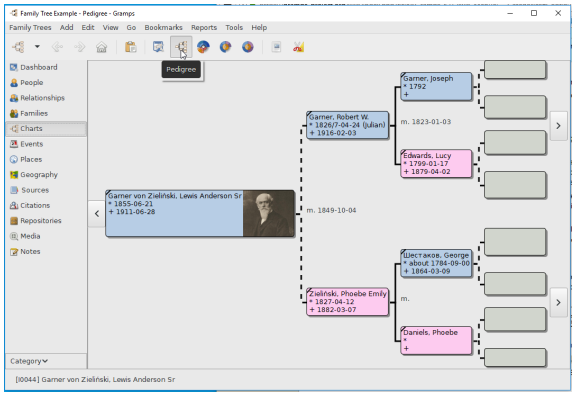


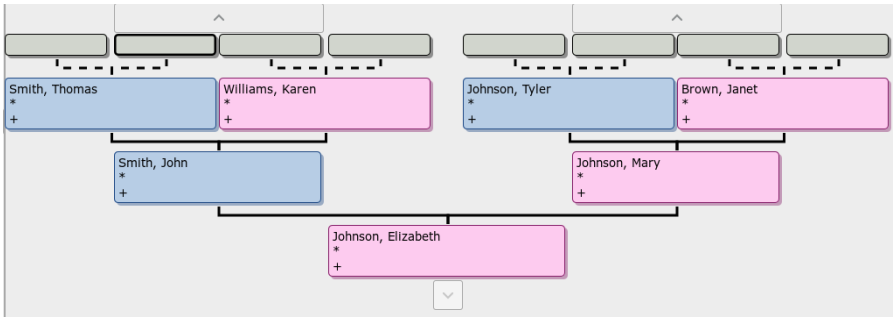
Figure 2.3: *Gramps* user interface [Allingham, 2020].

One of its main advantages is the abundance of graph types that it allows for, as well as the previously referred support for different add-ons, which may add more layouts and representation types to account for different use cases. At its default setting it supports interactive representation of vertical and horizontal trees, pedigree trees (Figs. 2.4a and 2.4b), as well as circular fan charts (Fig. 2.4c).

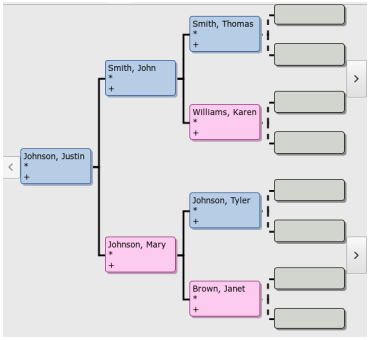
These types of graphs are usually enough for simple family trees, but whenever uncommon family links like children born out of incest, divorces and remarriages appear, the diagram is presented with duplication of people, which may confuse its users. This phenomenon will be referred to as node duplication.

Node duplication is the solution that many genealogy programs use for dealing with people that would be farther away in a representation like a hierarchical tree. A person from the tree that has a relation in a higher level cannot have another relation to someone below in the hierarchy. To solve this, the common person is duplicated so he can be in both levels on the graph at the same time. Figure 2.5 shows shows this problem, with the "Diego Smith" node being duplicated, which also leads to the duplication of his parent nodes. This problem is exacerbated in large family trees, where node duplication can occur frequently.

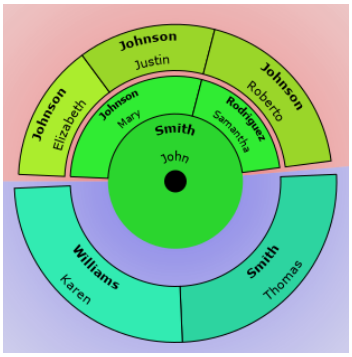
A representation such as the one illustrated in Fig. 2.6a uses a node-like connection rather than a strict tree, which allows better representation of the unusual family links. Having nodes instead of a traditional tree allows the representation of a double marriage to be presented without



(a) Vertical family tree.



(b) Horizontal family tree.



(c) Circular fan chart.

Figure 2.4: Family Tree Representations on *Gramps* [Allingham, 2020].

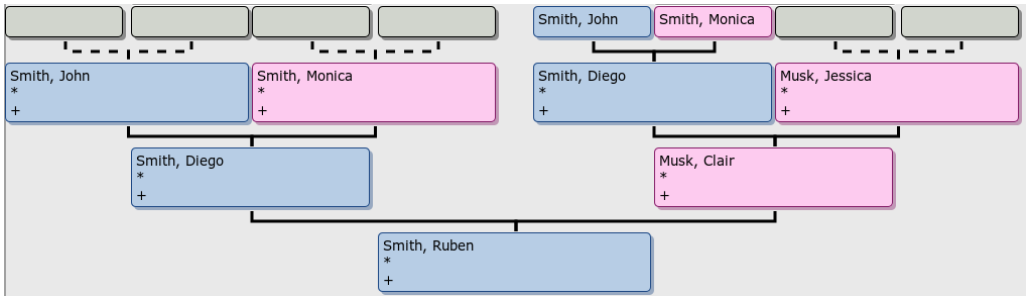


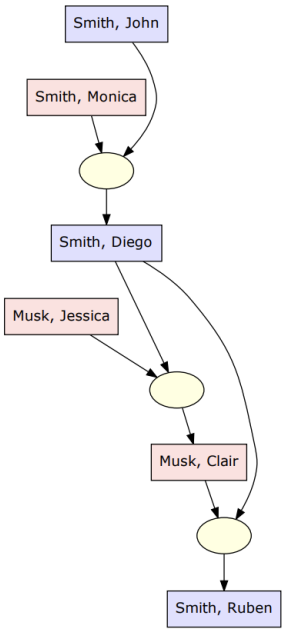
Figure 2.5: Node duplication example on *Gramps* [Allingham, 2020].

the need for node duplication. This representation is a feature only available when exporting a family tree in *Gramps* and as such it is static and restricts all interaction to panning and zooming.

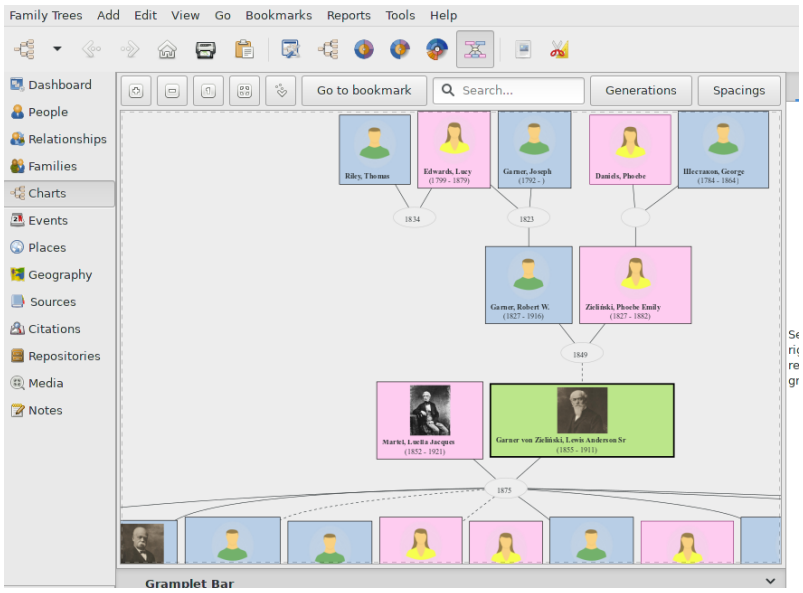
However, there are add-ons available that approach this issue. *Graph View* (Fig. 2.6b) is an add-on that creates a representation that allows for better interaction, similar to the default trees of *Gramps* whilst containing more information, similar to a descendant chart [Burton, 2020].

The *Graph View* add-on gives a more comprehensive view of the family trees as a whole, as well as how the different members of a family relate to each other. The differences in representation provided by this add-on provide information that would need to be gathered in a text-based menu in *Gramps*.

Further proving the importance of modularity in *Gramps* is the existence of completely different representation methods than what it usually provides. For example, if a user needs to see



(a) *Gramps* descendant chart.



(b) Graph View Add-on for *Gramps*.

Figure 2.6: Advanced representations on *Gramps* [Allingham, 2020].

a representation capable of displaying large amounts of data (up to several thousand individuals), they could install the Quilt add-on (Fig. 2.7). The add-on is inspired in *GeneaQuilts*, a visualisation technique developed by Bezerianos et al. [Bezerianos et al., 2010].

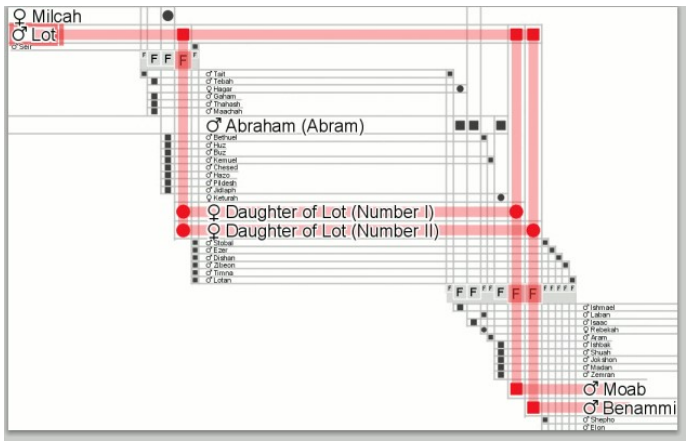


Figure 2.7: Quilt representation based on GeneaQuilts.

On the other hand, a user wishing to see how different people are positioned in a timeline, or if certain people are contemporary and how they relate to certain events in history, can install the *TimeNet* add-on (Fig. 2.8). Although this layout does not give much information regarding events, marriages and other data from the family tree, it excels in time-based tasks.

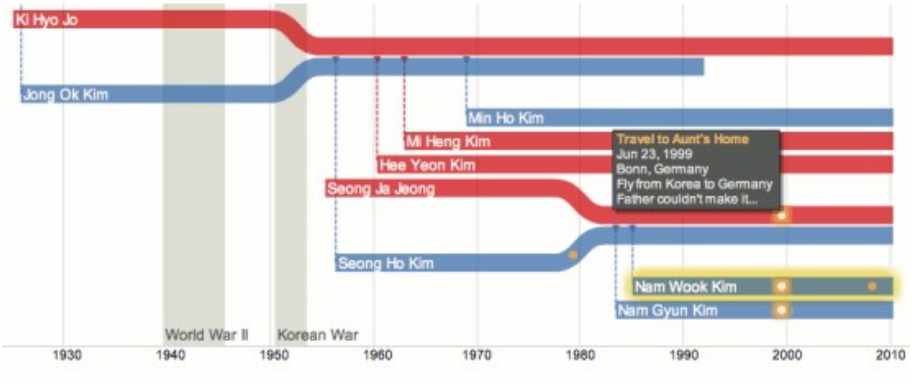


Figure 2.8: TimeNet representation based on Timeline trees..

2.3.2 Agelong

One other program used by the Genealogy community is *Agelong* (Fig. 2.9) [Genery Software, 2021].

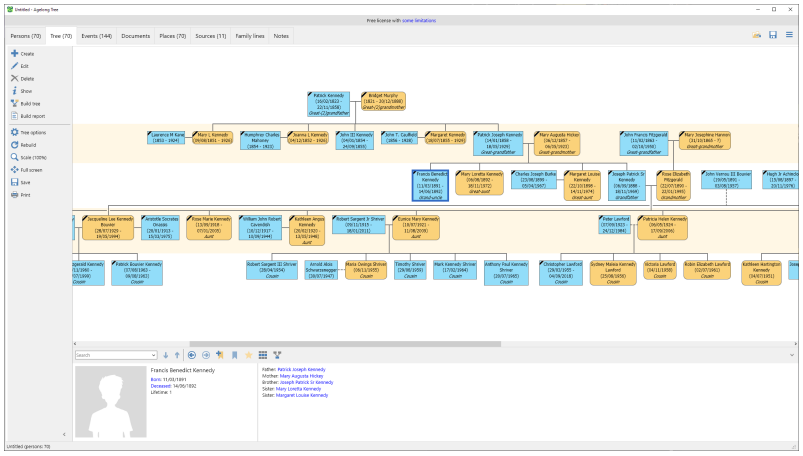


Figure 2.9: Agelong UI.

Like *Gramps*, *Agelong* is a fully-featured application with more than just graphical representations. It has several different ways to visualise the information of a family, from text-based methods grouping relevant information together to graphical representations of family trees. *Agelong* differs from other genealogy programs in its representation of a family tree due to the inclusion of a full family tree view (Fig. 2.10) instead of an abridged version.

Its representation is apt for most usual cases in family trees, and it avoids node duplication seen in some of the hierarchical tree representations (Fig. 2.11a), but has issues in the display of some edge cases such as people with more than one marriage in the context of a full tree (Fig. 2.11b).

In this specific case, the auto-generated layout positioned both the husband and ex-husband of "Janet Norton Lee" on the right side, with a line connecting to her descendants with the ex-husband "John Vernou". Having the representation done in this fashion creates a graph that can become confusing very rapidly. Additionally, this representation allows for interaction but it is fully static

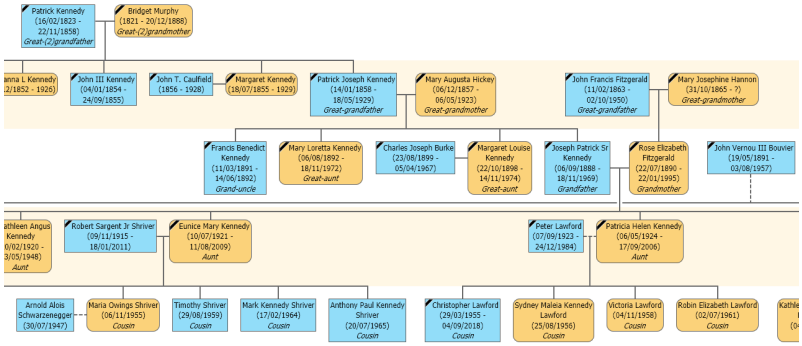
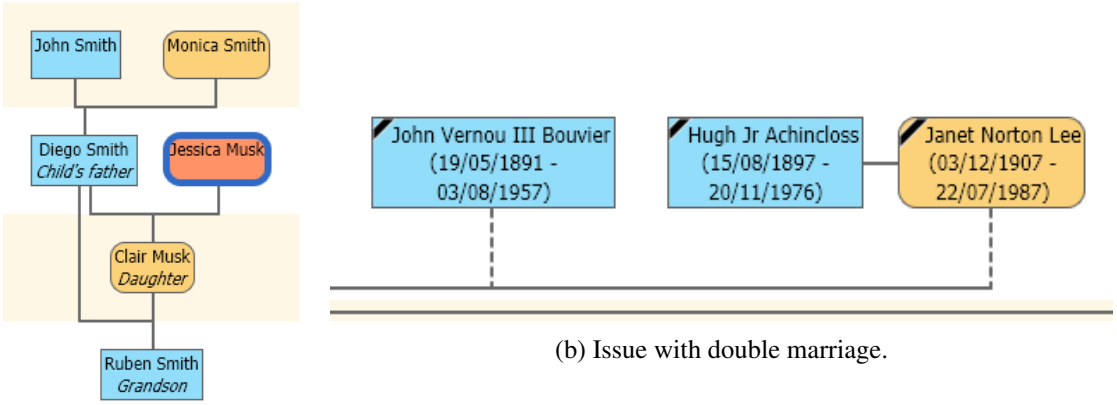


Figure 2.10: Tree representation without hidden people in *Agelong*.



(a) Uncommon relation.

(b) Issue with double marriage.

Figure 2.11: Agelong representations.

in its layout, not allowing users to re-position the family members as they see fit, meaning that a user is unable to fix a mistake done by the auto-layout.

2.3.3 Family.Show

Family.Show (Fig. 2.12) is a program developed with the set goal on attempting to achieve innovation in the representation of graphs while providing a good user interface experience.

It's a project started in 2007 and has since been discontinued in 2017 by the original developers[1]. As it is open-source, different developers may use it either as a basis for new platforms or to improve upon the existing one.

The user interface of *Family.Show* has several features that are interesting when compared to other genealogy programs. The rightmost bar offers most of the information pertaining to what a user has selected, and it allows editing of some data fields for each family member in a tree. Additionally, like other genealogy programs, it allows for direct search of all the family members, including their birth and death dates, from the side bar. There is also the inclusion of a slider on

[1] https://archive.codeplex.com/?p=familyshow

Figure 2.12: Family.Show UI.

the bottom left which allows for the highlighting of a specific year. By doing so, people that are not alive in that year are faded away and so are relationships that did not exist at that point. This simple tool is very interesting for the purpose of exploring contemporary family members and could provide better temporal awareness of relations and people in different historical events. The graphical representation of the family also presents some interesting implementation decisions compared to representations in other programs. It offers a more node-edge style of view of a family tree but has some issues with representation of bigger families (Fig. 2.13).

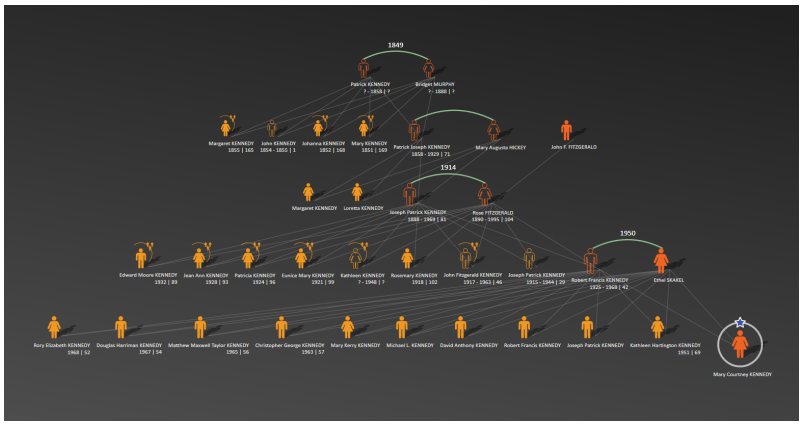
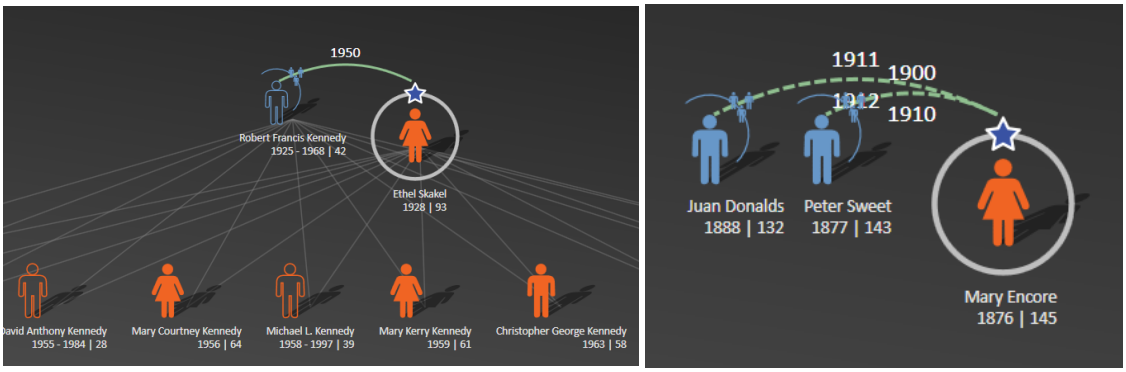


Figure 2.13: Big family tree in Family.Show.

Instead of representing all the members of a family, *Family.Show* often hides people, creating a tree that is smaller and easier to read but that lacks information. The hiding of the members is not made obvious when a family tree is first shown, thus making incorrect assumptions about the family more probable. Another issue with this style of representation is the overuse of connections between people and their descendants (Fig. 2.14a). By having both parents connected to each of their children, as the tree grows and lines cross more frequently, it becomes harder to follow these connections. This issue is avoided by other programs by the use of a relation node where the

parents and the children are connected.

One other issue with *Family.Show* is with the representation of more than one marriage/divorce (Fig. 2.14b). In this example "Mary Encore" has two divorces with different people, and because of the way it is displayed, a lot of the information is overlapping. Additionally, due to how *Family.Show* hides information to keep family trees simplified, one marriage between "Mary Encore" and "Peter Sweet" is completely absent from the graph and only the two divorces are shown. This behaviour differs from some other programs that instead display the most recent relation, which would be the marriage.



(a) Too many connections example. (b) Double Divorce.

Figure 2.14: Family.Show issues.

2.3.4 Family Echo

Family Echo[2] is an online web application developed for the representation of family trees (Fig. 2.15).

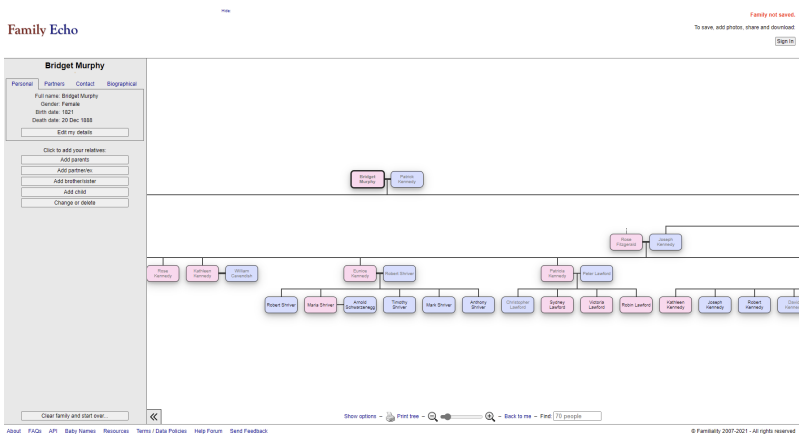
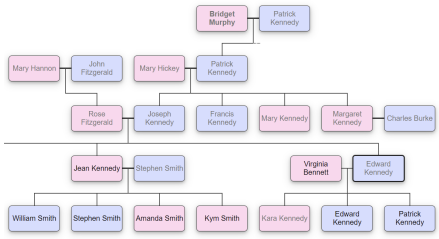


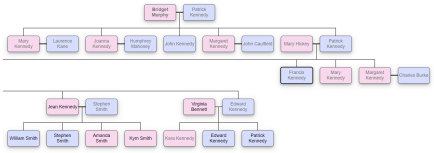
Figure 2.15: *Family Echo* UI.

[2] https://www.familyecho.com

It was developed by Gideo Greenspan in 2015 and remains a platform used by 100 million people[3] [Greenspan, 2015]. The graphical representation produced by this program is mostly a complete tree but, like many genealogy programs, it chooses to hide less related branches of the tree (Fig. 2.16).



(a) Hidden family.



(b) Real amount of children.

Figure 2.16: Family Tree Representations on *Family Echo*.

The members in the top of the graph have only two children in the first representation (Fig. 2.16a), but in reality they have 5 children (Fig. 2.16b). This can lead to some users mistakenly assuming the number of children if they are assessing the information from the representation alone. This is common practice but some programs like *Agelong* avoid doing so.

Another issue that arises from *Family Echo*'s graphs is the lack of representation of multiple marriages/divorces to the same person (Fig. 2.17). In this small sub-tree, there are three relations to be represented: Two divorces and one marriage. However, due to constraints in the way the lines between people are represented, only the two most recent relations can be displayed.

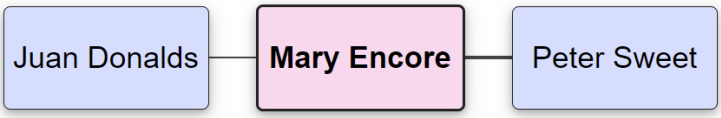


Figure 2.17: Hidded remarriage.

Furthermore, *FamilyEcho* does not support or allow for the creation and representation of uncommon marriages such as the ones between distant cousins and siblings.

2.3.5 Others

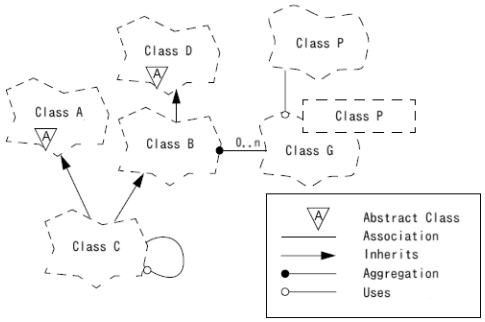
There are many other genealogy applications. Since many of them are paid or rely on subscriptions, they could not have been tested for the purpose of this work. Programs such as *FamilySearch*[4], *Ancestry*[5], *Family Tree Maker*[6], *Roots Magic*[7], etc. all provide different variants of

[3] http://www.gidgreen.com/
[4] https://www.familysearch.org
[5] https://www.ancestry.com
[6] https://www.mackiev.com/ftm/
[7] https://www.rootsmagic.com

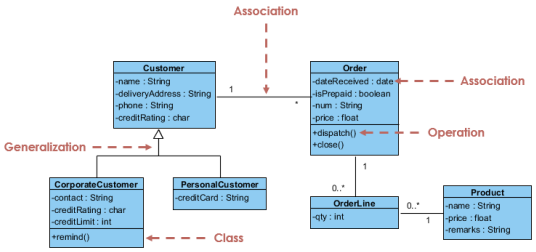
representations already discussed. The extent of these differences can't be further studied due to availability restrictions.

2.4 Existing UML class diagram representation platforms

The UML standard originated out of the necessity of managing the ever-growing complexity of systems. As the scale and scope of systems grew, so did the need for a language-agnostic system representation. The inception of this standard had basis in the unification of features from Booch method (Fig. 2.18a) with both the Object-Modeling Technique (OMT) and Object-Oriented Software Engineering (OOSE). It was a combined effort between several organisations that in 1997 created the UML modelling language version 1.0. Since then, the development has continued and as of the writing of this document is in version 2.5.1. There are thirteen different types of UML diagrams, each having its own array of use cases [Cook et al., 2017] [team, 2020] [Svinterikou and Theodoulidis, 1999]. For this work, the type of UML diagram to be explored is the UML Class diagram (Fig. 2.18b).



(a) Booch Method Diagram.



(b) UML class diagram.

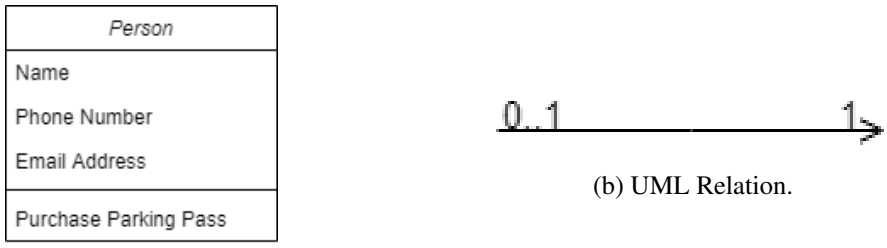
Figure 2.18: UML class diagram and its predecessor Booch Method Diagram.

The UML Class diagrams are mainly comprised of class tables (Fig. 2.19a) and relations (Fig. 2.19b). The class tables are objects divided into three main components: name, attributes and operations. Relations contain information of how a class interacts with other classes or itself and are divided into different categories: Inheritance, Association, Aggregation, Composition and Dependency.

When attempting to improve the representation and organisation of UML class diagrams, it is imperative that its defining features and rules are not forgotten or hidden. Respecting the rules imposed by the standard is essential.

Different types of UML class diagram editors have different types of audiences and thus, different levels of complexity. Simple diagram editors like *Draw.io*[8] enable users to make diagrams fast but do not enforce the UML class diagram standard, making them ideal for quick diagram

[8] https://app.diagrams.net



(a) UML Class.

(b) UML Relation.

Figure 2.19: UML class and relation example.

sketches, while more powerful and often expensive programs like *Enterprise Architect*[9] or *Visual Paradigm*[10] allow the creation of more complete diagrams with better enforcement of the UML class diagram standard. Considering that the differences between UML editors are not as significant as with genealogy programs, the following section is divided into professional tools and casual editors.

2.4.1 Professional Tools

Editors such as *Enterprise Architect*, *StartUML*[11] and *Visual Paradigm* (Figs. 2.20 and 2.21) are professional tools better suited for users seeking to create a complete diagrams following the UML class diagram standard.

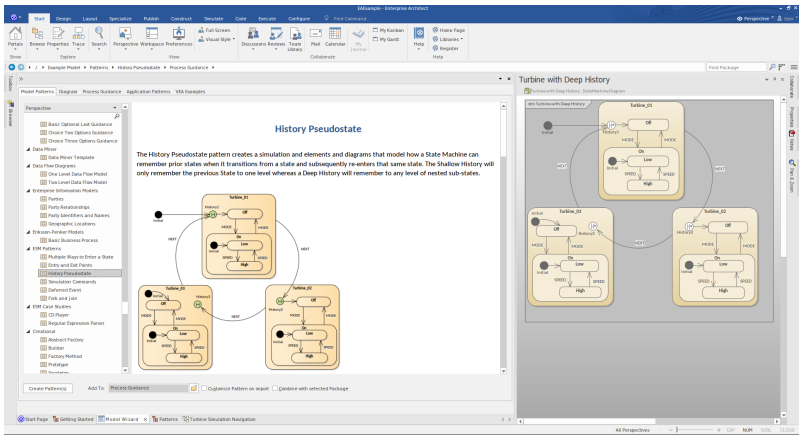


Figure 2.20: Enterprise Architect UI.

Regarding representation, these more complex editors offer powerful tools such as auto-layout based on different algorithms (Fig. 2.22) granting users the ability to automatically rearrange their diagrams in an optimal way.

Additionally, these editors allow the storage of diagrams in their proprietary file formats and allow importing and exporting of the diagram's data in the standard XML Metadata Interchange

[9] https://sparxsystems.com/products/ea/
[10] https://www.visual-paradigm.com
[11] https://staruml.io

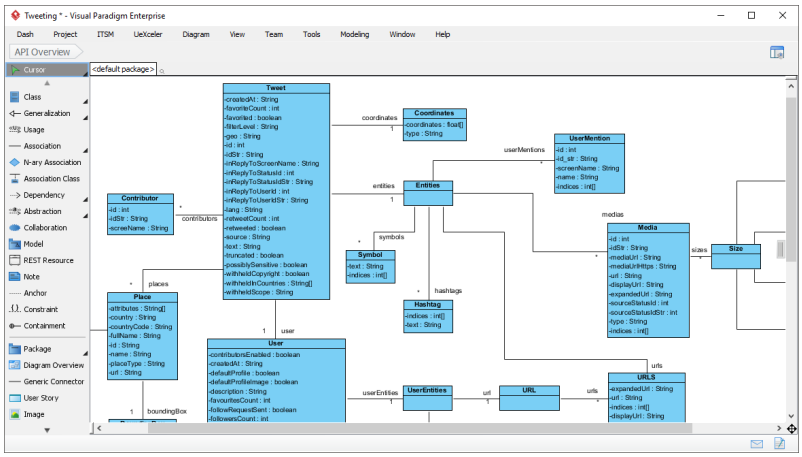


Figure 2.21: Visual Paradigm UI.

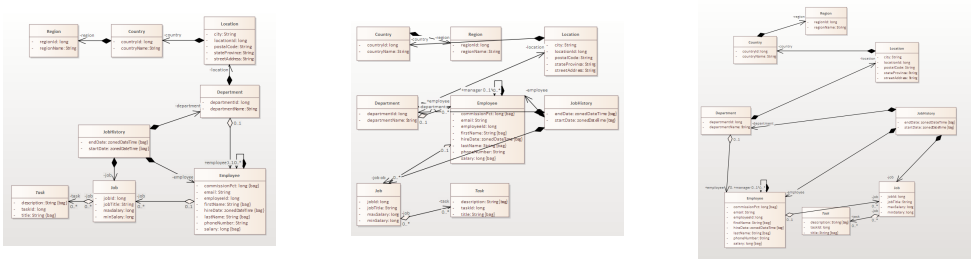


Figure 2.22: Different Enterprise Architect layouts.

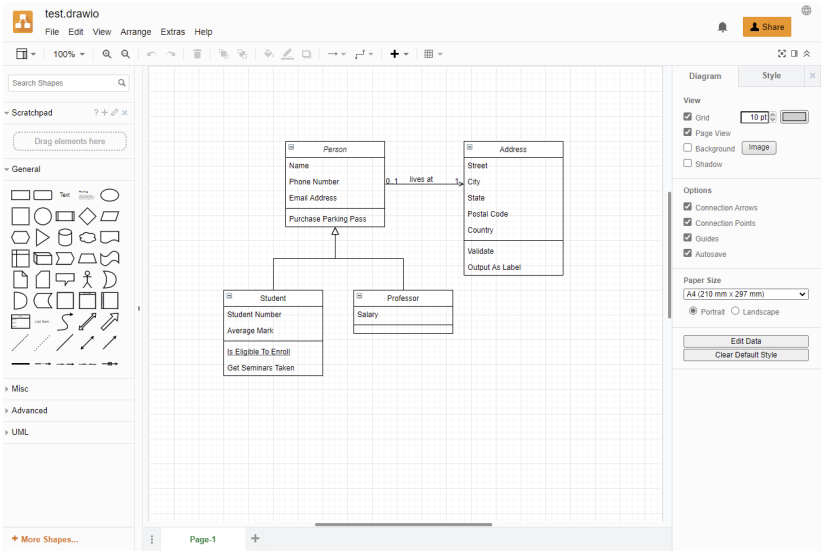
(XMI) file format. Importing an XMI file does not create an automatic layout, requiring the user to place the diagram class elements manually. This is due to the separation between the model of a UML class diagram and the diagram representation itself. To see a representation of the diagram, the user must first manually place all the UML classes and then position them or use an automated layout, if available.

These current implementations are very capable and grant users many different avenues to produce quality diagrams. Unfortunately, they do not offer meaningful and intuitive tools for diagram exploration and interaction, as they are mostly focused on the creation of static diagrams and not in how a user can interact and extract information.

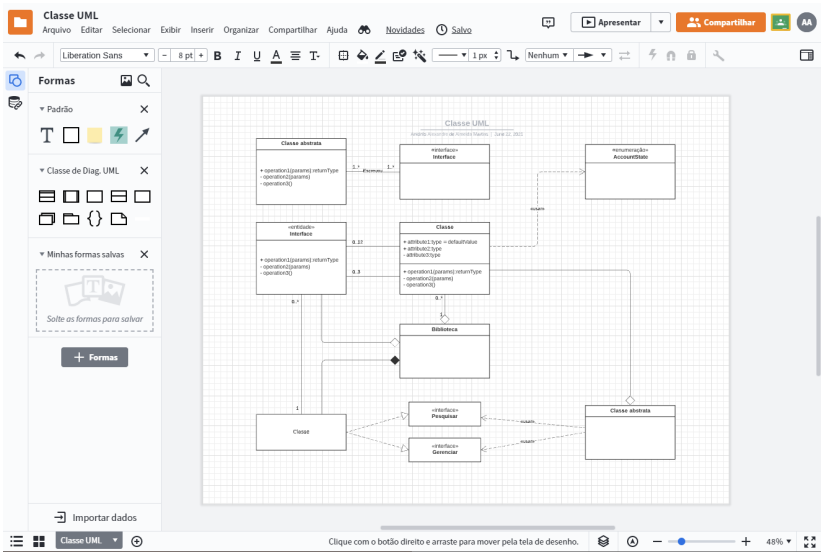
2.4.2 Casual editors

Contrary to professional UML class diagram editors, there are more casual and often free editors that allow the creation of UML class diagrams and more. Many of them are available as web applications, giving them flexibility and compatibility with multiple devices. *Draw.io*, *Lucidchart*[12] (Figs. 2.23a and 2.23b) and web versions of professional programs such as *Visual Paradigm Online*[13] are examples of these simpler editors.

[12] https://www.lucidchart.com
[13] https://online.visual-paradigm.com



(a) Draw.io.



(b) Lucidchart.

Figure 2.23: Different web based editors.

These editors are often multi-purpose applications with very loose restrictions in regards to following guidelines for specific diagrams. Both *Draw.io* and *Lucidchart* allow the placement of objects from different types of graphs, giving users the option to create hybrid diagrams that do not follow any particular subset of rules.

Additionally, the generated diagram can only be stored in a proprietary file format, exported as an image or saved in generic graphic file types. The hybrid nature of these diagrams also restricts the file importing capabilities, not being able to use file types for specific diagram types, such as UML.

Being straightforward applications, without restrictions in their representations, allows for these editors to have much simpler interfaces and to create graphs, like UML class diagrams,

at a much faster pace. However, the layout capabilities of these programs is also limited, with less features than their more professional counterparts, something especially apparent in *Visual Paradigm Online* when compared to the full version of *Visual Paradigm*.

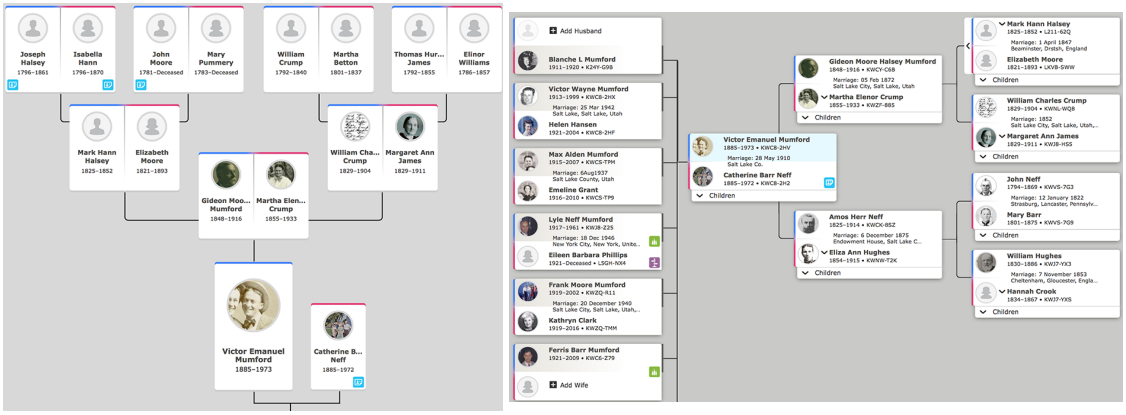
2.5 Existing graph representations and Layouts

As demonstrated in the previous section, there are many ways to represent different types of information when using graphical representations. For the purpose of this work, the platform implemented must be able to represent the different graphs with appropriate layouts and organisation algorithms for both Family Trees and UML Class diagrams. This section discusses the different graph types commonly used to represent this type of data, as well as their advantages and drawbacks.

2.5.1 Hierarchical Trees

Horizontal and Vertical trees (Fig. 2.24) are widely used to represent family tree graphs and allow users to get a good overview of the data displayed. Choosing between both orientations can help the representation of the graph on different screen aspect ratios. Programs like *FamilySearch* and *Gramps* allow users to pick between both orientations whilst keeping the same information. Hierarchical tree representations excel in the representation of data that makes use of hierarchy, as in data arranged in different levels, such as family trees and UML class diagrams.

Some programs such as *FamilySearch* and *Gramps* choose to use the simpler variant of the tree, referred to as pedigree tree (Fig. 2.24a). This is ideal for exploration of the descendants and ancestors of a singular individual on a family tree, having the tree always end on a singular person with branches for their ancestors. The main issue with this representation is the lack of scalability and flexibility it offers.



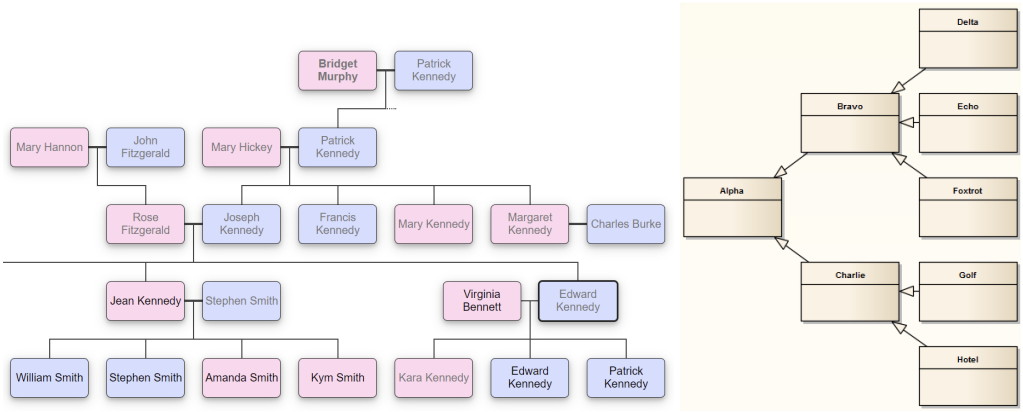
(a) Vertical Tree.

(b) Horizontal Tree.

Figure 2.24: Family Tree types by *FamilySearch*.

Other programs like *Family Echo* use hierarchical trees to produce representations that are less limited than pedigree trees. These trees often show all the available data, and chose to hide elements only if they are not as closely related to a selected element or if showing them would clutter the representation (Fig. 2.25a).

Enterprise Architect also provides a similar tree representation named *Digraph* with a focus on ordering the elements of the UML class graph with hierarchy in mind (Fig. 2.25b).



(a) Full tree in *Family Echo*.

(b) Digraph in *Enterprise Architect*.

Figure 2.25: Hierarchy focused representations.

These full scope representations provide good results for both of the required data types and as such was a strong consideration for the work.

Like horizontal trees, indented trees (Fig. 2.26) offer a good visualisation of the information if the data is appropriate but, similar to pedigree tree representations, node duplication can occur, making analysis more difficult.

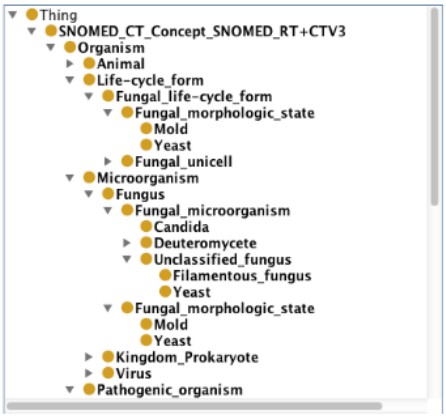


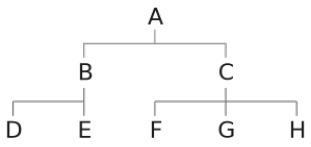
Figure 2.26: Indented Tree [Fu et al., 2013].

A study evaluated how users perceived information from indented trees and graphs and concluded that while most users in the study found the indented trees more appealing and less confusing, when the data became more complex, users were more prone to making mistakes. In-

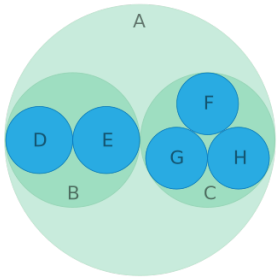
stead, users would make less mistakes when using graph representation if allowed more time [Fu et al., 2013]. The study suggests that indented trees are more suited for list-checking activities and that graphs offer better representation for overviews, a finding that can be very useful for this work.

2.5.2 Circular Treemaps

Circular Treemaps grant a new representation for traditional trees that may be more apt for certain types of information. They keep the hierarchy of the tree representation unchanged but, when increasing the number of nodes of a tree, they can become more confusing than regular vertical and horizontal trees. This representation type allows for the implementation of simple exploration methods such as zooming into different nodes of the tree and the size of the nodes can be used to convey information pertinent to the diagram type. Figure 2.27 demonstrates how a circular treemap (Fig. 2.27b) is essentially just a different representation of a regular hierarchical tree diagram (Fig. 2.27a).



(a) Tree representation.



(b) Tree map representation.

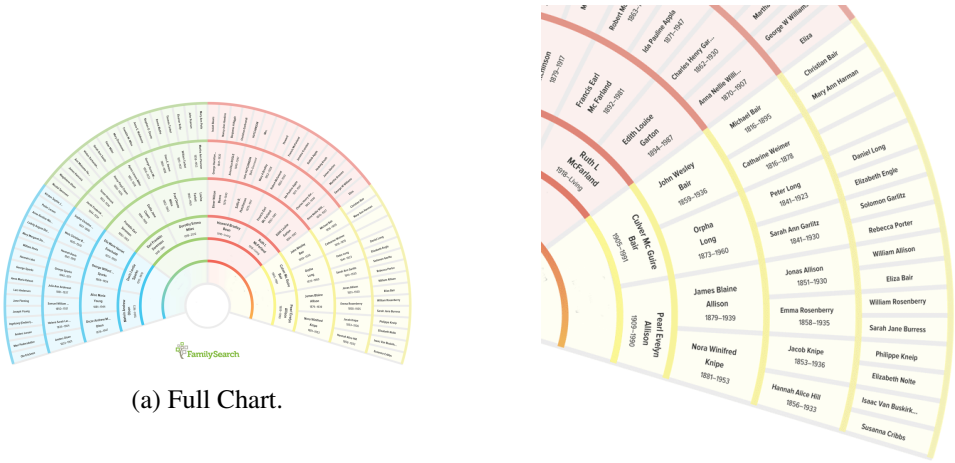
Figure 2.27: Circular Treemap example.

While this representation offers an appealing variant to the normal tree representation, it is not suitable for family tree or UML class diagrams due to the fact that when representing multiple levels of a tree, it becomes hard to differentiate between parent nodes and their descendants. Additionally, the representation does not lend itself for manipulation and editing, making it very restrictive for exploration and does not help fix the main issues that traditional family tree representations have, such as node duplication.

2.5.3 Circular Fan chart

Circular fan charts are a widely adopted method for representing information that has an organised hierarchy. This particular kind of diagram is often used for representing segments of family trees, particularly when exploring the lineage of a specific family member. *FamilySearch* is a service that allows the creation of such family trees (Fig. 2.28) and uses it to give a quick overview of the graph.

As noted previously, other programs like *Gramps* and *FamilySearch* also make use of this representation method for great effect, leveraging the layout for use on exploration. It is in smaller



(a) Full Chart.

(b) Segment of the Chart.

Figure 2.28: Circular Fan Chart.

branches of family trees that this representation excels, as representing an entire family tree can overload the graph or be impossible. Although it is a powerful implementation for family trees, it carries limitations that come from the need for hierarchy, so that using it for different types of data, such as UML class diagrams, could be unfeasible or unnecessarily complicated.

Furthermore, this representation, similarly to hierarchical trees, can suffer from node duplication.

2.5.4 Force-directed

The representation of graphs using force directed algorithms, also known as spring embedded algorithms, dates as far back as 1963. Since then, force directed algorithms have evolved and formed different variations, with each one offering advantages and disadvantages over the others [Kobourov, 2012].

Vizster (Fig. 2.29) makes use of force-directed graphs to represent links between users in a social network. In this representation, the data elements are represented in a network layout using a spring-embedding algorithm in which the tension between nodes reflects the relation between them. Two related nodes will have less tension between each other than two unrelated nodes. This allows the related nodes to be closer together, making relevant data stand out [Heer and Boyd, 2005].

One of the biggest advantages of this format is allowing the data to be automatically arranged in groups based on the relation between the nodes in a graph, ideal for the display of an overview of a given data type, be it family tree or UML class diagram.

However, the layout does not lend itself for precise exploration as the nodes in the groups formed by the force-directed algorithm do not follow any strict rules for hierarchy or organisation. For that, different layout should be considered, one that better allows the representation of the relations between nodes and their relevance.

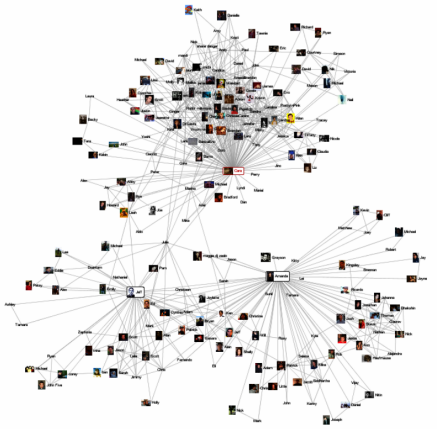


Figure 2.29: Vizster by Heer et al. [Heer and Boyd, 2005].

2.5.5 Radial and Force Directed

The possibility of creating a layout out of other existing ones should not be ignored. By combining different techniques, new representation methods can be formed with advantages pertaining to the used layouts. When creating an alternative representation for family trees, Keller et al. developed a representation that combined different already existing layouts, radial layout and force directed layout (Fig. 2.30).

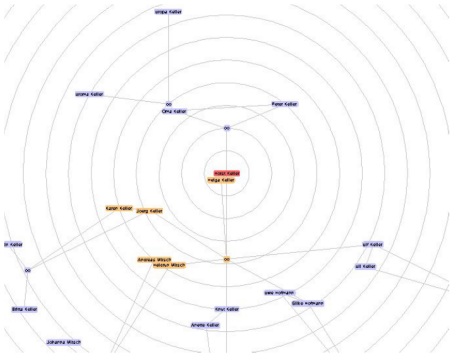


Figure 2.30: Keller et al. diagram [Keller et al., 2011].

To organise nodes the authors make use of the distance from the node to the centre node as a representation of time. This allows users to better acknowledge time differences between people in the family tree. The usage of force directed algorithms in this representation allowed the graphs to remain readable and less cluttered [Keller et al., 2011]. This study provides insight on how a mixed approach using different algorithms and representations could be used in this work as an apt new representation that meets its needs. Additionally, the use of time as a factor for the representation can prove useful in the creation of a family tree layout.

2.5.6 GeneaQuilts

As referred to previously, the *GeneaQuilts* method of visualisation of family trees was build with the goal of achieving a good representation method for family trees with a very large number of individuals [Bezerianos et al., 2010]. Unlike with family trees that rely on node-link representations, *GeneaQuilts* takes a different approach by using a matrix based visualisation (Fig. 2.31).

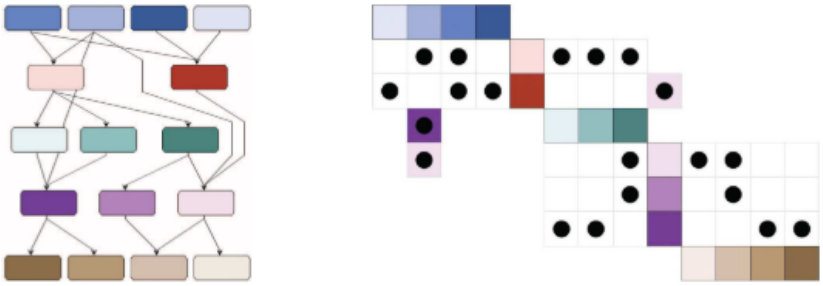


Figure 2.31: Quilts by [Bezerianos et al., 2010].

The figure 2.31 shows how a node-link graph would translate to a *GeneaQuilts* matrix and with it, it is possible to see how the use in family trees is possible. The main advantage of using this kind of representation is allowing for the display of big families without the drawback of overly cluttered graphs. It's an interesting approach for family trees but its type of representation does not translate well into UML class diagrams, as with that type of information, there are more types of data to be displayed in relations, as well as classes.

2.5.7 Comparison

Considering both of the types of data to be used, it is important to choose the representation types that better suit the goals of this work.

Indented trees produce representations that can be viable for simple diagrams, but cannot present all of the characteristics of more complex diagrams such as UML class diagrams, and struggle with difficult cases in family trees.

Representations such as circular tree maps, circular fan charts, pedigree trees and *GeneaQuilts* are all able to represent family trees, and can provide information about relations between elements, but they are not ideal for producing diagrams such as UML class diagrams. The complexity in the UML class relations would be limited by restrictions in the representation and the information of each class would be relegated to sub-menus or sidebars, making it unnecessarily complex.

From all the studied representations, the layouts that proved usable in both family trees and UML class diagrams are reliant on either a node-edge approach, like force-directed or radial graphs, or a hierarchical tree representation. These layouts are also the ones that manage to avoid having to resort to node duplication.

2.6 Interactions/Inputs

To create an apt representation platform for both of the intended diagrams, having good visualisation is not enough. As stated previously, one of the most important factors in graph interpretation that is often left forgotten is interaction. The importance of interaction in InfoVis systems is universally recognised but little is made in the way of improving it in programs [Lee et al., 2012]. This observation is proven by most of the already discussed platforms, since they use the traditional desktop user interfaces that rely only on cursor and keyboard. These types of user interface rely on the same principal, named *WIMP* (window, icon, menu, pointing device), and its by keeping that approach that they have not evolved into more interactivity friendly systems. For this work, interaction is a very important factor, and as such different input methods where studied and evaluated in their availability, usability and what additional features they can provide.

2.6.1 Cursor

Compared to other input methods, the cursor has been present across a multitude of operating systems and devices thought the years.

The first computer mouse was created circa 1960 and saw its first true use in 1970, when it was used to control *Xerox* computers. Through the years they became more and more popular to the point that they are almost always supported on graphical operating systems [Pang, 2002]. This led to the widespread adoption of the use of cursor as a method of interaction with systems and the creation of alternative cursor control methods.

Nowadays, a cursor is controlled by a computer mouse, trackpad, trackball mouse or touchscreens, all with varying types of precision and learning curves (Fig. 2.32). Their ease of use usually depends on the quality of the input device and the software implementation being used.

One of the main advantages of using a cursor as an input device is the ability to select and drag screen elements, in this case when interacting with graphs, as well as facilitating interaction with UI elements. Additionally, along with a cursor comes the wheel functionality that enables the usage of functions such as zooming and menu navigation. When a device controlling a cursor does not have a physical wheel, it can replace it with touch controls on the mouse or as gestures in the case of a trackpad or touchscreen. Most devices sold today, be it desktop PCs, laptop PCs, tablet computers and even mobile phones are capable of using a cursor as a method of navigating through their system and applications [Snehi, 2006].

The usage of a cursor for the proposed solution is a strong consideration as current implementations of graph representation/editing such as *Gramps*, *Visual Paradigm* and *Enterprise Architect* all rely on it to work to some extent [Allingham, 2020] [Sparks, 2020]. Additionally, since cursor controlling devices are widely available, choosing this method will widen the variety of support devices.

[1] https://commons.wikimedia.org/wiki/File:3-Tasten-Maus/Microsoft.jpg
[2] https://commons.wikimedia.org/wiki/File:Macbook/touchpad.jpg
[3] https://commons.wikimedia.org/wiki/File:Wireless-trackman-mouse.jpg

(a) An optical mouse [14].

(b) A touchpad [15].

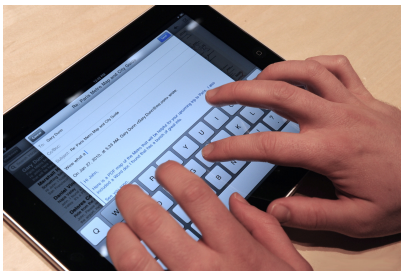
(c) A trackball mouse [16].

Figure 2.32: Different types of pointer controllers.

2.6.2 Keyboard

The computer keyboard was born out of the necessity for text inputs for early computers. Computer keyboards go as far back as 1956, when a keyboard was used for the first time in a computer, the *Wirlwind* [Everett, 1980]. In the 1970s, computer keyboards were starting to be adopted by several brands to create personal computers. This continued until today, where most conventional computers, laptops and desktops, use keyboards (Fig. 2.33a) as the primary text input method. Advances in portability and touchscreens allowed many devices to forgo physical keyboards, instead opting for touch/gesture-based operation or resorting to digital "Soft Keyboards" (Fig. 2.33b), keyboards on a touchscreen display [Bellis, 2020]. As an input, keyboards are essential for text introduction, a requirement when implementing features such as searching and keyboard shortcuts.

(a) A physical keyboard [17].



(b) A software keyboard [18].

Figure 2.33: Different types of keyboards.

Software keyboards may offer convenience as they cut size from the device, but they also introduce many issues when compared to physical ones. Most virtual keyboards vary greatly in what they can and can't do. For example, most hardware keyboards are capable of using shortcuts such as key combinations for activating certain commands in a program. Software keyboards on tablet computers and smartphones usually do not allow this kind of interaction and are restricted to just text input. This inconvenience can be overcome in different ways such as the usage of touch gestures and on screen buttons to replace the keyboard shortcuts. In addition to these issues, software keyboards vary greatly depending on what maker or brand implemented it. Unlike hardware

[4] https://commons.wikimedia.org/wiki/File:QWERTY/keyboard.jpg
[5] https://commons.wikimedia.org/wiki/File:Apple/iPad/Event03.jpg

keyboards that are mostly standard in functions and layout, software keyboards have additional complications and limitations that originate from their host device [Nakagawa and Uwano, 2011].

Computer keyboards are available on most devices destined for work or leisure and they have been available for a long time, making them a very good candidate to use as an input method for this work.

2.6.3 Touch

The introduction of touch devices to control computers was on 1965 and has been improved over time with different types of touch capable devices [Johnson, 1965] [Johnson, 1967]. The rise in adoption of smartphones and tablet computers led to the creation of new ways of using touch-capable devices, other than just controlling a pointer [Schmidt and Churchill, 2012]. By using different gestures through multi-touch (using more than one touch point at a time), stylus pens or even pressure sensitivity, touch devices became more user friendly and intuitive.

In a work by Basheri et al. that analysed how the usage of multi-touch impacted the collaboration between users when manipulating UML diagrams (Fig. 2.34), touch-enabled devices proved to provide a lot of flexibility and usability. Additionally, it was found that using the multi-touch approach was beneficial to the participation between users and encouraged "parallel participative design", making multi-touch a method to be considered [Basheri et al., 2012] [Basheri et al., 2013].

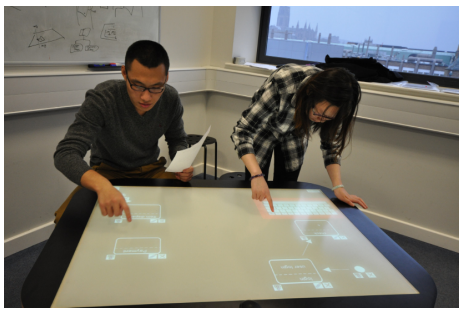


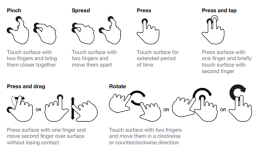
Figure 2.34: Collaborative UML table using MT-CollabUML.

These benefits shown are unfortunately overshadowed by inherent problems of touch technology on a collaborative level. For example, the audience for family tree diagrams is very likely to not have a multi-touch table and the collaboration benefits that come from it may not be necessary or useful. Additionally, targeting such a niche use case would limit the usability of the platform

Smaller touch devices such as tablets were considered as candidates for the implemented solution as they allow for input with different types of touch methods: Touch, Multi-Touch, Stylus or any combination of the three (Fig. 2.35). Stylus and non multi-touch are considered the same for the purpose of this work, as the increased precision afforded by the stylus does not give any appreciable value to diagram manipulation.

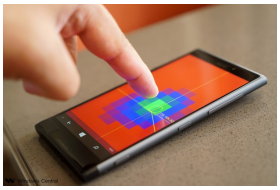
Multi-touch on the other hand can give the user avenues for more intuitive actions and allow for more actions with less input steps, for example, a pinch gesture for zoom can be simpler than

using buttons for the same effect. A gesture based system would allow more screen space for the graphical representation.



(a) Multi-touch gestures [19].

(b) Stylus touch screen [20].



(c) Pressure sensitivity [21].

Figure 2.35: Different types of touch capabilities.

Although at this point most mobile devices are capable of touch and many allow for multi-touch, the intended work is not suited for such small devices such as smartphones, as the perception of information can be greatly affected by the size of the display used.

2.6.4 Virtual Reality (VR)

Apart form conventional input methods, there are other more specialised tools that could provide interesting methods of visualisation and interaction with diagrams. One of such methods is virtual reality (VR), that in displaying the diagram data in a 3D environment, creates the need for a dedicated gesture-based input approach. Huang et al. worked upon this idea of implementing the visualisation of data graphs and how to interact with them in a virtual reality environment. This work concluded that the usage of gestures improved usability when compared with traditional mouse inputs when in VR [Huang et al., 2017].

Compared to other methods, VR is a recent technology and as such has the potential of providing advantages in both display and input methods due to how it differs from older input peripherals on computers. However, these advantages are not without drawbacks. A survey by *Statistica* (Fig. 2.36) examined two hundred industry specialists in "Obstacles to mass adoption of virtual reality (VR) technologies Q1 2019" and concluded that VR is still a very new technology, with major hurdles to overcome, mainly being the hardware and lack of supporting software. This funnels into costumers being more reluctant to acquiring a VR capable device, which in turn further reduces the userbase of the technology [Holst, 2021].

In addition to these obstacles, VR has been shown to produce adverse physical effects such as motion sickness in many individuals. A work by Munafo et al. concluded through two experiments that an alarming number of users experienced motion sickens after using VR for fifteen minutes. In the first experiment, users were tested with an Occulus Rift Dk-2 (Second revision of the development kit of the Occulus Rift) headset while seated and playing a game that used head

[6] https://cdn.thedesigninspiration.com/wp-content/uploads/2012/10/gesture-icons-002.png

[7] https://ibcdn.canaltech.com.br/w660i90/galaxy-note-20-ultra-s-pen.jpeg

[8] https://www.windowscentral.com/sites/wpcentral.com/files/styles/w830_wm_blb/public/field/image/2016/07/mclaren-5.jpg?itok=1n1OFdfK

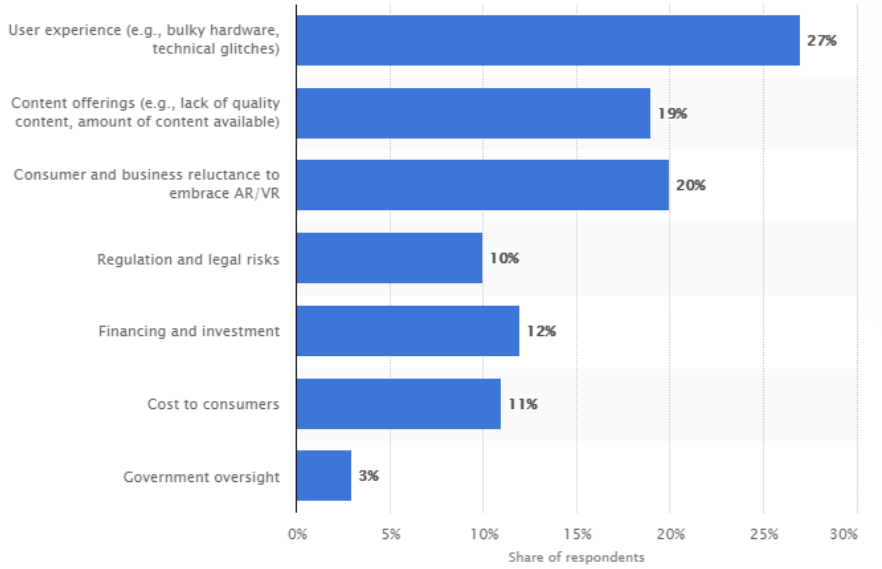


Figure 2.36: Survey by *Statistica* [Holst, 2021].

motion to guide a marble through a maze. The conclusion of this test was that 22% of the subjects reported motion sickness. In the second experiment, the users were required to again play a game while seated, with the goal of navigating through an environment of hallways and rooms. This resulted on 56% of the users reporting motion sickness after fifteen minutes [Munafo et al., 2017].

A different study by Nguyen et al. compared how differences on the users and the type of VR hardware they possess can lead to different levels of motion sickness. From the sample of almost 300 users, 57.8% reported to experience some kind of motion sickness (Fig. 2.37). The work further explored how age and sex relate to susceptibility to motion sickness, concluding that older users and woman were more inclined to suffering motion sickness from VR. Motion sickness was also related to the devices used, with information concluding that users with cheaper VR equipment had more reported cases of motion sickness between the test group [Nguyen et al., 2020]

The combined weight of the limitations has critically left VR with a slow development and a slow growth. Developing for this kind of platform would be counter productive as the goal of the work is to be widely used by as many people as possible, be it home users viewing family trees or professionals analysing UML diagrams.

2.6.5 Comparison

Different input methods provide different advantages and disadvantages depending on the use case. Table 2.1 below provides a comparison of the previously discussed input devices:

Apart from the different capabilities, the availability of input devices weighs on the decision of what devices to implement. A survey by *Statista* ran from 2010 to 2020 saw a growth on laptop computers shipments and reduction of tablet computers and desktop PCs throughout the years

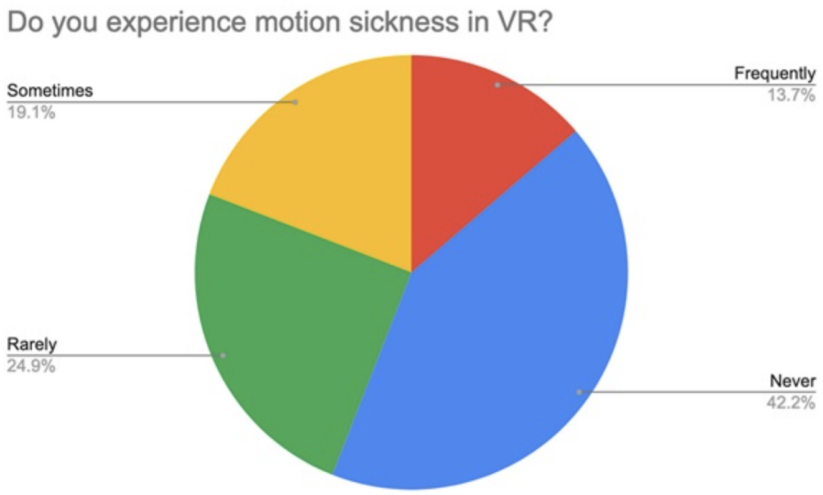


Figure 2.37: Percentage of users with motion sickness by [Nguyen et al., 2020].

Table 2.1: Feature comparison between input devices

	Click/Tap	Selection	Drag-drop	Pan	Zoom	Write text	Shortcuts
Mouse	✓	✓	✓	✓	✓		
Keyboard		✓[1]		✓	✓	✓	✓
Touch/Stylus	✓	✓	✓	✓	✓[2]	✓[3]	
Multitouch	✓	✓	✓	✓	✓	✓[3]	

Notes:
1 - Although keyboards do not provide the user with a cursor to select items, they are capable of doing so through the usage of shortcuts such as the TAB, to navigate through selectable elements in the User Interface (UI).
2 - Devices with a touchscreen capable of only recognising one point at a time are still able to Zoom through the usage of onscreen buttons.
3 - Touch and Multi-touch screens are capable of displaying software keyboards that allow for the writing of text. They do not have the same capabilities as regular keyboards due to the lack of shortcuts.

(Fig. 2.38). Additionally, it predicted that laptops sales would continue to grow as the other two would dwindle slowly.

As shown by the survey, laptops and desktop PCs hold 65% of the units shipped. The possibility of implementation of touchscreen control should not be completely invalidated as laptops have begun implementing touchscreens and tablet computers hold a significant margin of units, about 35%. [Alsop, 2020]

All in all, mouse and keyboard are the ideal candidates for implementation, not only by accounting for the wide availability of laptops and desktop PCs, as discussed previously, but also their ease of use for a wide audience.

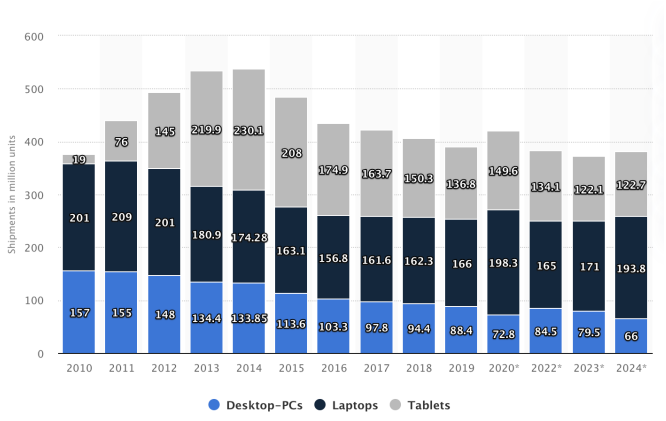


Figure 2.38: Shipment forecast of tablets, laptops and desktop PCs worldwide from 2010 to 2024 by [Alsop, 2020].

Chapter 3

Problem and Proposed Solution

This chapter describes common problems found in both graphic representations as well as how some programs try to solve them. Decisions made for input devices and algorithms are also discussed, and several representation platforms are compared in their capabilities to represent the data and to support needed features.

3.1 Common Problems in Family Trees

Genealogy programs have many use cases, most of them centred on giving users information about the family tree they are observing, while allowing for some customisation of those trees. These are seemingly easy tasks but different family trees can present challenging representation scenarios for these programs.

3.1.1 Big families

The primary issue genealogy programs struggle with is the representation of large numbers of family members in a complete family tree while maintaining clarity. Issues such as overlapping relations, duplication of individuals and convoluted information arise with large amounts of data and, as such, must be circumvented in order to create a good user experience.

One way of solving this issue without the hassle of managing a lot of data is hiding less related information. *Family.Show* uses this method in an attempt to simplify the graph and avoiding too much information displayed at once. Although this makes the diagrams appear simpler and easier to read, the hidden information can result in users misinterpreting the data. The example family tree *The Kennedy Family.GED* is composed by 70 people with their respective relations and additional information. With *Family.Show*, it is not possible to represent the full tree as it always opts for representing partial trees based on the selected person, even when navigating through the family (Fig. 3.1). This, in conjunction with not informing the user about the hidden family members worsens the usability of the platform.

Gramps is also not able to represent the full tree in a dynamic way. Users are given the options to use fan charts (Fig. 3.2c) or limited vertical/horizontal trees, sometimes referred to as pedigree

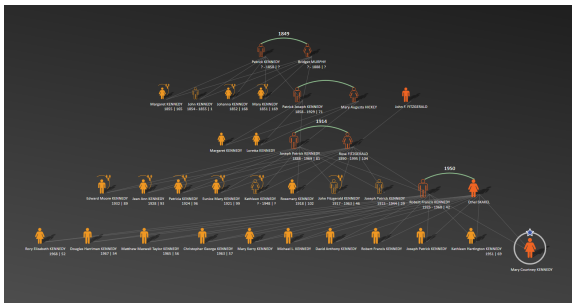
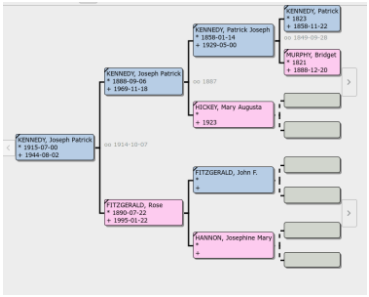


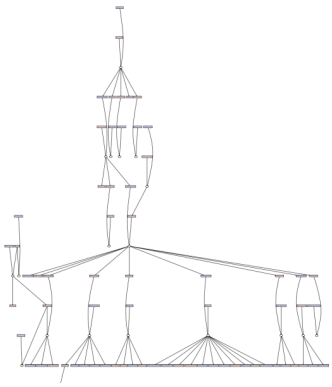
Figure 3.1: Kennedy family tree represented in *Family.Show*.

trees, (Fig. 3.2a) that converge on one person, methods that can have some interactivity, but hide a lot of information and suffer from the same issues as regular hierarchical trees.

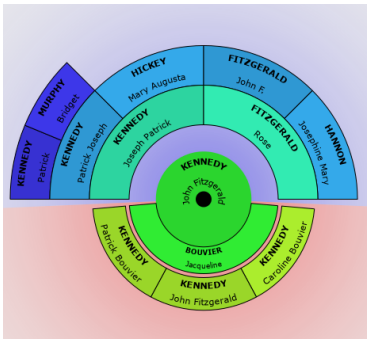
The one method *Gramps* has to fully represent a tree is through exporting a family tree as an image in a PDF file (Fig. 3.2b). This method gives the full view of a family tree but the resulting diagram is static, completely removing any possibility of interaction and graph manipulation.



(a) Gramps Tree



(b) Gramps Export



(c) Gramps Fan Chart

Figure 3.2: Gramps family representations

Other programs like *Roots Magic* and *Family Echo* can create partial trees, but much like *Family.Show*, they hide information that might not be as related and don't provide an option for a full family representation. *Agelong* is the only program tested that was able to create a complete family tree representation while remaining interactive (Figs. 3.3).

3.1.2 Unusual connections

Another issue that genealogy programs face is the representation of unusual relations between people. Traditionally, family trees have simple connections such as marriage, divorce, non-martital partnerships and biological or adoptive child that are easy to represent. The issue arises when attempting to represent less common cases such as:

- **Multiple relations between the same people** — Most programs opt to show this information either on the connection between the married people or in an information page or

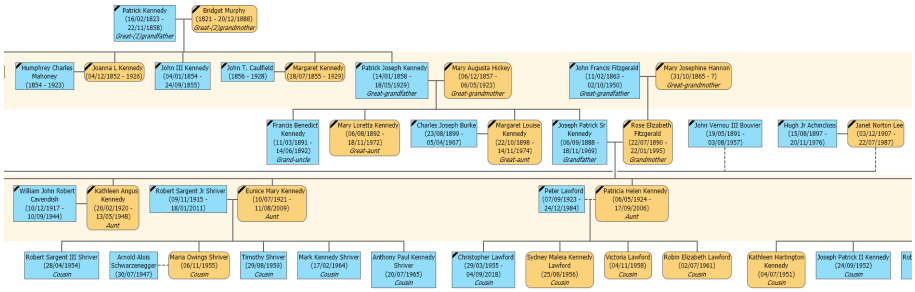


Figure 3.3: Agelong full tree view

sidebar when selecting either the married people or the marriage connection. The use of information page or sidebar allows for the display of additional information such as marriage or divorce dates, as well as locations or proof of marriage.

- **Multiple simultaneous relations to different people** — Usually, genealogy programs solve this problem by using node duplication. By duplicating people in a tree/diagram, they programs can represent them in multiple spots at the same time. This method, although effective, contributes to cluttering of the tree and may lead to misinterpretation of information.

- **Relations between unusual people (Cousins, Brothers, Intergenerational couples)** — The representation of this kind of relation greatly depends on the program used but is mostly done through node duplication.

- **Relations with multiple descendants that also have relations with descendants** — When representing the descendants of a family it is important to keep the children and their respective sub-families from overlapping. A common practice from genealogy programs to avoid this issue is hiding the families from the siblings that are currently not selected.

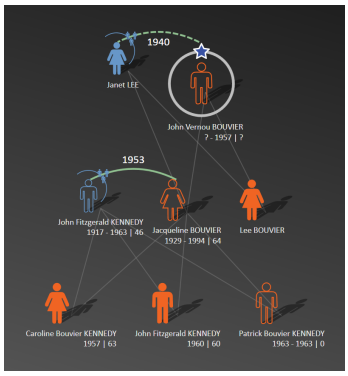
- **Relations with multiple descendants that also have relations with ancestors** — Like with descendants, it is important that, when representing ancestors of the children of a family, the nodes don't overlap and are positioned in a place that makes sense for the family represented. The most common practice by genealogy programs is to hide the ancestors of the children unless they are selected.

- **Descendant with both adoptive and biological parents** — When representing a child with both a biological and a adoptive families, it is important to position the families to avoid overlapping whilst keeping the rest of the family tree organised.

These issues can appear isolated or in combination, making representation even more complex.

Solving the issue of multiple marriages can be as simple as just displaying the most recent marriage or divorce and displaying the remaining information about the older relationships elsewhere, in a sidebar or popup. This is the approach taken by *Family.Show* (Fig. 3.4a) as it is the simplest way of showing only the most relevant information.

Websites such as and *Gramps*, *FamilySearch* (Fig. 3.4b) and *Ancestry* also chose to simplify the family trees, opting to only show a chosen marriage at a time, resulting in information being hidden in sub-menus.



(a) *Family.Show* multiple marriages.

(b) *FamilySearch* multiple marriages.

All in all, while this behaviour can sometimes provide cleaner looking family trees, it also creates the possibility of the user misinterpreting the data, as the hidden family tree branches make the trees misleading.

3.1.3 Approachability/Ease of use

One common characteristic across all tested genealogy programs is how complex the user interface can become due to the usage of the classic WIMP interaction style. Simple interactions such as viewing the entire family tree, adding a new family member and seeing all information about a family member can become unnecessarily convoluted due to the existence of many features hidden in sub-menus [Walny et al., 2012].

Programs like *FamilySearch* allow users to search for family members, but when representing people with more than one relation, it chooses to hide every other relation in sub-menus. Furthermore, accessing most of the information about a person requires the user to leave the family tree that he is viewing and to re-open the graph to continue his exploration.

Other applications like *Family.Show* often chose to hide family members to avoid cluttering the graph. In practice, this results in graphs that appear less complex but require more effort to navigate. Additionally, information about marriages and divorces is often hidden in sub-menus or completely absent.

Some of the complexity of the UI of these programs can be justified due to the amount of features they possess, but often are due to the age of the programs and the reliance on older architectures.

3.2 Common Problems in UML class diagrams

UML class diagrams are complex types of diagrams that follow a set of rules defined by the OMG UML class diagram standard.

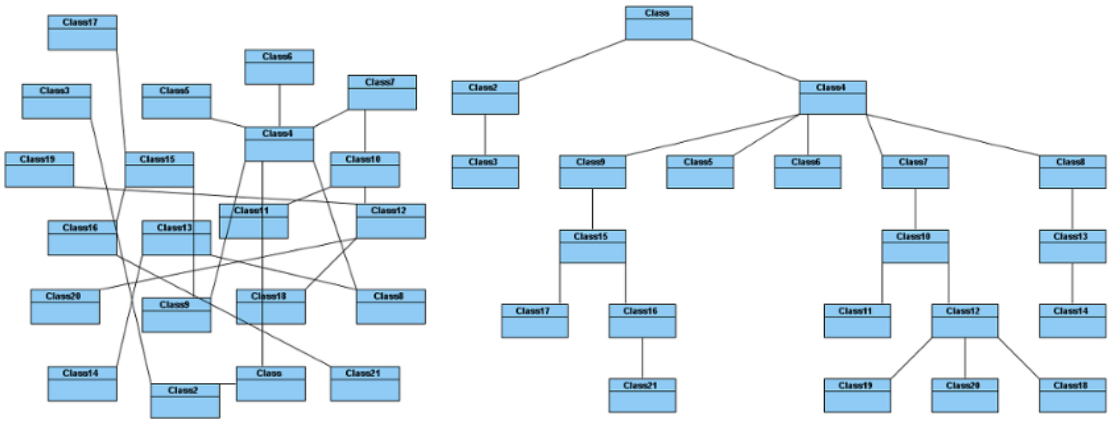
The complexity of the diagrams varies with the complexity of the systems they are meant to represent and different programs have different ways of dealing with this complexity. As the diagram grows, problems start to arise depending on the visualisation platform that is being used.

3.2.1 Layout

In UML class diagrams with a sizeable amount of classes and other information it becomes inevitable that some of the relations overlap. Not only can this overlapping happen between relations, but also between nodes or a mix of both.

By placing a diagram manually, it is possible to avoid some of the issues, however this practice is unfeasible for bigger diagrams. Instead, layout algorithms must be used and, depending on these, results can be varied.

Powerful editors like *Visual Paradigm* allow users to pick from a list of available layouts or to create a custom one with configurable parameters. For example, *Visual Paradigm* can organise an otherwise confusing diagram (Fig. 3.5a) into a readable representation (Fig. 3.5b)

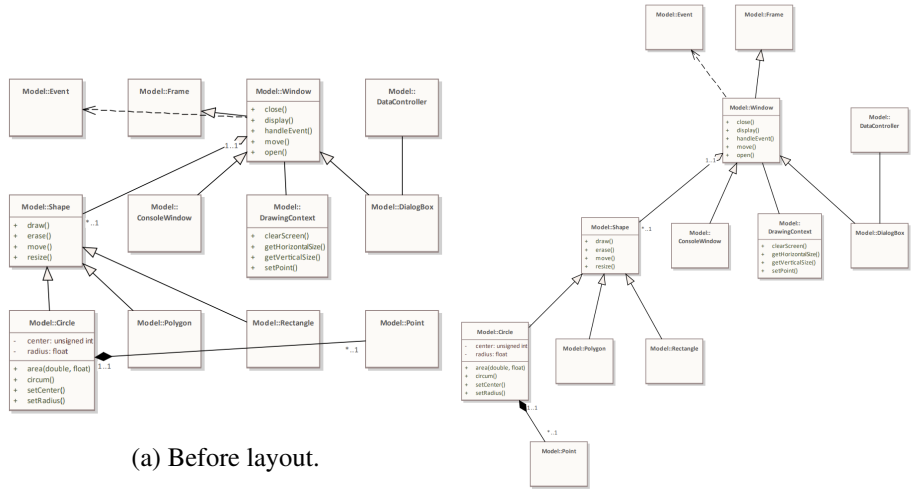


(a) Before layout.

(b) After layout.

Figure 3.5: Visual-Paradigm Tree Layout.

This type of automated layout can be simple or complex, depending on what information is accounted for when organising the classes. In the previous example, the classes represented are only composed of simple associations, so there is no need for any hierarchy or specific ordering. When ordering UML class diagrams with hierarchic relations in mind, more advanced algorithms need to be employed. As previously referred, *Enterprise Architect* uses a algorithm named *digraph* (Directed Graph) that allows UML diagrams to be arranged with generalisation hierarchy well represented whilst keeping crossed relations at a minimum (Fig. 3.6).



(a) Before layout.

(b) After layout.

Figure 3.6: Enterprise Architect Digraph (Directed Graph) Layout.

However, both these examples are relatively simple diagrams. When it comes to representing a large number of classes and relations, the automatic layouts begin to struggle and the only possible way to represent all the classes is through overlapping (Fig. 3.7).

Figure 3.7: Visual-Paradigm overlapping

It becomes impossible to represent a UML class diagram with such a large amount of interconnected classes so, in order to solve this issue, additional representation techniques must be used in

addition to normal layout algorithms.

For this work, it is important to create layouts that are able to reproduce readable graphs in both simple and complex cases. This includes the capability of representing hierarchy and other UML characteristics if they are available in the UML model.

3.2.2 Readability

One characteristic of UML class diagrams that is important to keep well represented is the complexity of each class and its relations. Each class is composed of several components: class name, attributes, operations and stereotypes. Relations are also composed of several different parts that need to be well represented: relation name, multiplicity and roles on each end of the relation, symbols and line style representing the type of relation, etc. Apart from classes and relations, there are also other important components such as class packages, comments and associative classes.

To produce a coherent and complete representation, all these elements must remain readable after a layout is produced.

3.3 Inputs

The intended solution is meant to be as readily available as possible. For that, the inputs used must be commonly available in the intended audience devices. Furthermore, they need to perform adequately and reliably to guarantee that a user can interact with the solution properly and intuitively.

Most available implementations of family tree and UML class diagrams representation and editing rely on the usage of a cursor and keyboard exclusively. Programs such as *Enterprise Architect*, *Visual Paradigm* and *StarUML* all use a cursor and keyboard for their inputs. This is also true for most installable family tree editors like *Gramps*, *Agelong* and *Roots Magic* and web based editors like *Ancestry*, *Family Search* and *Family Echo*.

This continuous use of cursor and keyboard can be attributed to how their workflow was moulded, technologies available and accessibility of constraints. The cursor is used to interact with the user interface and to select and move elements, while the keyboard is used for shortcuts and text input. This does not restrict these programs from using other inputs like touchscreens, as all current operating systems allow for the manipulation of the cursor through touch and for text input via virtual keyboards. Unfortunately, this does mean that these programs do not leverage the advantages brought by technologies such as multi-touch touchscreens and gesture support.

As stated in a previous chapter, recent device sales also show that most devices commonly sold contain mouse and keyboard as their main input devices making them the ideal candidate for implementation on this work. The existence of touchscreen devices should not be overlooked, but due to the lesser number of available devices, it should not take precedence over a cursor and keyboard.

3.4 Layout/Algorithm

This work intends to provide a new approach to representing two different diagram types, Family Trees and UML diagrams. For that to be possible, the implemented layouts must be capable of representing both types of diagrams with accuracy and adequate interactivity.

As discussed in a previous chapter, data can greatly influence how the graph is represented and organised. The way a user interprets the data displayed is dependent on how it is displayed and how he can interact with it [Lee et al., 2018] [Yi et al., 2007].

Previous works tackling visualisation of family trees made use of different layouts, from simple circular fan charts and trees to node-networks organised through radial graphs or force directed algorithms [Keller et al., 2011]. UML class diagrams do not differ as much in implementations and tend to always follow the conventional UML standard with different programs using different layouts. *Enterprise Architect*, *Visual Paradigm* and *StarUML* all have different customisation for the organisation of the graphs, most allowing the placement and movement of the class tables manually with the possibility of executing algorithms to further organise the class tables.

For this work, the usage of a combination of algorithms and layouts can provide a representation for each of the graph types that will keep interaction, usability and readability on the forefront.

3.5 File Format

To make the intended implementation viable and usable, compatibility with existing data must be granted within the solution. It is impossible to grant compatibility with all data and filetypes, so choosing the filetypes which encompass the broadest amount of data available for each of the diagrams is needed.

3.5.1 Family Trees

The focus in preservation of genealogical history has led to the creation of various different approaches to archiving the data. There are many file formats used to store this information and they all have different advantages and disadvantages.

GEDCOM, Genealogical Data Communication(Fig. 3.8) is one of the oldest methods for storing family information when building a family tree or any related graph. It was first created in 1984 and has remained the *de facto* standard file type ever since. The format has suffered several iterations over the years with the latest version being 5.5.1, released on November 2019[1].

The data model supported by *GEDCOM* is lineage-linked, making it so that all the information is based on the believed reality, meaning families and individuals. This method contrasts evidence-based models which have the data structured according to supporting evidence.

The *GEDCOM* format brings many advantages due to its wide and continuous implementation such as many adaptations to different programming languages and the ability of converting many other file formats into it. Its longevity and usage as a standard makes it one of the most widely

[1] https://www.familysearch.org/wiki/en/GEDCOM

```
0 HEAD
1 SOUR PAF
2 NAME Personal Ancestral File
2 VERS 5.0
1 DATE 30 NOV 2000
1 GEDC
2 VERS 5.5
2 FORM LINEAGE-LINKED
1 CHAR ANSEL
1 SUBM @U1@
0 @I1@ INDI
1 NAME John /Smith/
1 SEX M
1 FAMS @F1@
0 @I2@ INDI
1 NAME Elizabeth /Stansfield/
1 SEX F
1 FAMS @F1@
0 @I3@ INDI
1 NAME James /Smith/
1 SEX M
1 FAMC @F1@
0 @F1@ FAM
1 HUSB @I1@
1 WIFE @I2@
1 MARR
1 CHIL @I3@
0 @U1@ SUBM
1 NAME Submitter
0 TRLR
```

Figure 3.8: *GEDCOM* file format example (.ged)

used formats, leading to much of the available data being stored in the format and its shortcomings being widely documented. Being an older standard also brings forward some disadvantages such as being hard to understand as a human, not including multimedia files and not fully respecting the *Genealogy Proof Standard* [Search, 2021].

GEDCOM X (Fig. 3.9) was announced by *FamilySearch* in 2012 as possible successor for *GEDCOM*, bringing improvements to the format. The main goal with its creation is to fix long standing issues with the original *GEDCOM*, such as not having a well-defined source model, the inability of storing multimedia files and not fully supporting the *Genealogical Proof Standard*, whilst keeping it an open source approach to minimise the need of proprietary standards.

```
<gedcomx xmlns="http://gedcomx.org/v1">

  <person id="P1">
    <source description="#SD1"/>
    <name>...</name>
  </person>

  <person id="P2">
    <source description="#SD2"/>
    <name>...</name>
  </person>

  <relationship>
    <person1 resouce="#P1"/>
    <person2 resource="#P2"/>
  </relationship>

  <sourceDescription id="SD1">
    <citation>...</citation>
  </sourceDescription>

  <sourceDescription id="SD2">
    <citation>...</citation>
  </sourceDescription>

</gedcomx>
```

Figure 3.9: *GEDCOM* file format example (.ged)

One main advantage of *GEDCOM X* for the purpose of this work is the implementations offered by FamilySearch that allow it to be serialised into Extensible Markup Language (XML) or JavaScript Object Notation (JSON) file formats. These formats are widely used and have good parsing implementations readily available [Richards Franklin, 2012]. Additionally, having good documentation and a well-defined source model can provide better development grounds for the implementation.

It is unfortunately not as widely adopted as *GEDCOM*, as it is more recent and as such much of the data would need to be converted before usage. To increase the adoption of the new format, *FamilySearch* provides the means to convert information from *GEDCOM* to *GEDCOM X* in the form of a *Java* parser.

Gramps uses a proprietary implementation extending XML named Gramps XML (Fig. 3.10), with the extension .gramps. This implementation is part of a greater file packaged named Portable

Gramps XML Package, with the extension .gpkg.

```
<person handle="_K8WJQCWGEQBNZ9EFJ6" change="1185438865" id="I0110">
  <gender>M</gender>
  <name type="Birth Name">
    <first>Christopher Arthur</first>
    <surname>Warner</surname>
  </name>
  <eventref hlink="_a5af0eb7cb96a818267" role="Primary"/>
  <eventref hlink="_a5af0eb7cca543f2664" role="Primary"/>
  <childof hlink="_9HTJQCJOU25SSV3SM"/>
  <citationref hlink="_c140d2641f710f04af6"/>
</person>
```

Figure 3.10: Gramps XML example (.gramps)

This implementation shares many similarities to *GEDCOM X* in that it was created due to the need of a better data system with more information compared to *GEDCOM*. It predates *GEDCOM X* and is used by programs such as *Gramps*, *PHPGedView*[2] and *Betty*[3] [Allingham, 2020].

One different type of file accepted by many genealogy programs is CSV files. This file format is often compatible with genealogy programs due to its long standing as a method of archiving data. There are many available options for converting CSV files into *GEDCOM* or Gramps file types.

Due to the lack of readily available data in the *GEDCOM X* format and having the majority of programs allowing for importing and exporting *GEDCOM*, this file type was chosen for implementation.

3.5.2 UML Diagrams

The UML standard was created in 1994 and was adopted by the Object Management Group (OMG) as a standard. The OMG has managed the UML standard since then. In 2005 the International Organization for Standardization (ISO) published UML as an ISO standard.

Like with family trees, there are many different types of formats used to store the information contained in a UML class diagram. These formats vary greatly in both the amount of information stored and the compatibility between programs.

Fully featured editors such as *Enterprise Architect*, *Visual Paradigm* and *StarUML* store their diagram information in proprietary files, allowing them to save additional information as they require, but reducing compatibility between themselves.

To pass information across different editors, a singular and standardised file type is required, the XML Metadata Interchange (*XMI*) format (Fig. 3.11).

The format was created by OMG with the intent of allowing the storage of the metadata of object diagrams and given that it was made a standard, it was adopted by most of the popular offerings of UML diagram editors.

Included in the *XMI* files is the information that makes up all the classes in the original graph and their components as well as their relations. In its standard variant, it does not contain any

[2] http://phpgedview.sourceforge.net
[3] https://pypi.org/project/betty/

```
<packagedElement xmi:type="uml:Class" xmi:id="_0iCzELieEeW4ip1mZlCqPg" name="Location">
  <ownedAttribute xmi:type="uml:Property" xmi:id="_0iCzEbieEeW4ip1mZlCqPg" name="country" visibility="private" type="_0i
  <ownedAttribute xmi:type="uml:Property" xmi:id="_0iCzErieEeW4ip1mZlCqPg" name="locationId" visibility="private" type="
  <ownedAttribute xmi:type="uml:Property" xmi:id="_0iCzE7ieEeW4ip1mZlCqPg" name="streetAddress" visibility="private">
    <type xmi:type="uml:PrimitiveType" href="http://www.omg.org/spec/UML/20110701/PrimitiveTypes.xmi#String"/>
  </ownedAttribute>
  <ownedAttribute xmi:type="uml:Property" xmi:id="_0iCzFLieEeW4ip1mZlCqPg" name="postalCode" visibility="private">
    <type xmi:type="uml:PrimitiveType" href="http://www.omg.org/spec/UML/20110701/PrimitiveTypes.xmi#String"/>
  </ownedAttribute>
  <ownedAttribute xmi:type="uml:Property" xmi:id="_0iCzFbieEeW4ip1mZlCqPg" name="city" visibility="private">
    <type xmi:type="uml:PrimitiveType" href="http://www.omg.org/spec/UML/20110701/PrimitiveTypes.xmi#String"/>
  </ownedAttribute>
  <ownedAttribute xmi:type="uml:Property" xmi:id="_0iCzFrieEeW4ip1mZlCqPg" name="stateProvince" visibility="private">
    <type xmi:type="uml:PrimitiveType" href="http://www.omg.org/spec/UML/20110701/PrimitiveTypes.xmi#String"/>
  </ownedAttribute>
</packagedElement>
```

Figure 3.11: *XMI* file example

information regarding the positioning of elements in a diagram nor any specific elements such as colouring or any sort of customisation.

Due to the omission of positions of the class diagram elements, several editors opt for the usage of their own proprietary file types or they add information that is extra to the standard in the form of *XMI* tags.

One such example is the editor *Visual Paradigm*, which chooses to include extra information in its *XMI* files to help customise its graph representation when importing a file into the program. This extra information is only usable by the program but does not interfere with other programs importing it.

As the format *XMI* was created by OMG, the creators of the UML class diagram standard, and has broad support from many editors, it was chosen as the format to be supported in the platform.

3.6 Representation platforms

The creation of a solution capable of granting a user the graph visualisation, interaction and compatibility with existing data requires the selection of a platform with sufficient capabilities for implementation. For the implementation to be efficient, the multiple libraries considered must be compatible with each other to minimise possible conflicts. The chosen libraries must also consider the previously alluded to components:

- **Inputs** — The platform chosen must be compatible with at least the basic inputs mouse and keyboard. Any other input methods, while not necessary, are welcome as it can expand the usability of the solution.

- **Layout/Algorithm** — Selecting a platform and libraries that support the needed layouts and algorithms, while allowing the implementation of custom variants of them, is essential for the correct development of the work.

- **File Format** — The usage of specific file formats for data importation for each of the diagram types requires the existence of tools that read those formats or a platform that allows the creation of such tool.

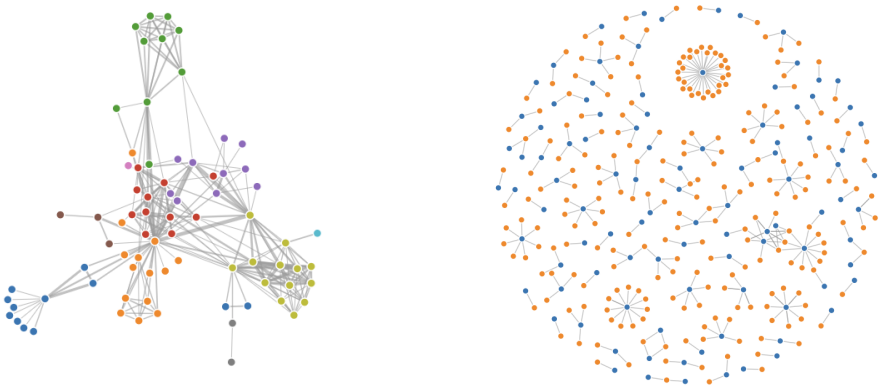
To guarantee proper implementation, the chosen platform must be open-source and be well documented, as well as being properly maintained and kept up to date.

3.6.1 D3.js

Data-Driven Documents, known as D3.js, is a powerful *JavaScript* (JS) library for representation and interaction of graphs. The library is based on HyperText Markup Language (HTML), Canvas and Scalable Vector Graphics (SVG) making it an ideal platform for a web-based implementation of the solution. Its development started mid 2011 by the same team responsible for *Protovis*, Mike Bostock and Jeff Heer of the Stanford Visualization Group and is presented as a successor to that framework [Bostock and Heer, 2009].

Jeff Heer was also responsible for the creation of *Prefuse*, a *Java* software tool for creation of interactive graphs, that had its support ended in 2007 in favour of Protovis[4] and later D3.js [Heer et al., 2005].

D3.js is considered one of the most complete and powerful representations for graphs and is popular among data scientists and data analysts. The framework allows for the implementation of different types of graphs, including several types of node-based representations such as Force-Directed Graphs (Fig. 3.12) and Radial Graphs.



(a) Force-Directed Graph.

(b) Disjoint Force-Directed Graph.

Figure 3.12: Force-Directed Graphs in D3.js.

Additionally, the library is still receiving updates and has extensive documentation and tutorials to aid implementation. Its reliance on web technologies allows it to remain available cross platform and its focus on minimal overhead further extends the number of devices capable of using it.

[4] https://mbostock.github.io/protovis/

3.6.2 Processing

The development for *Processing* started in 2001[5] as an effort to promote software literacy and to aid different groups such as designers, coders, film makers and educators. The platform is divided into three different modules:

- **Processing** — A *Java* implementation of *Processing*. Started as a tool for software sketch booking and developed into more powerful software for developers and educators alike with features making it ideal for prototyping and education. It is open source and compatible with Windows, Linux and macOS, making it suitable for the purpose of this work. Figure 3.13 shows how the library can be used to produce graph representations like force-directed graphs.

- **P5.js** — Like the *Java* implementation, P5.js has the same goals but applies them in a *JavaScript* approach. The main differentiating qualities that separate it from *Java* Processing are the focus on drawing capabilities on browsers, given by the use of add-on libraries that facilitate interaction with HTML and device hardware in a web environment. Similar to Processing, it is still in active development and has proper documentation to aid implementations.

- **Processing.py** — Although a Python implementation of processing exists, it is still being developed, and as such lacks complete documentation, and is not as feature complete as the other *Java* and *JavaScript* counterparts, making it not as suitable for implementation compared to them.

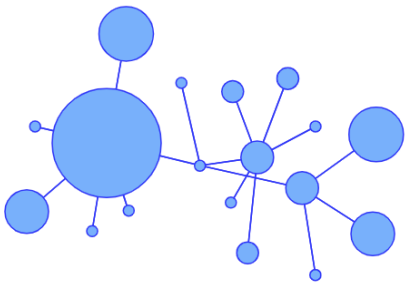


Figure 3.13: Force-directed graph in Processing.

As a platform, *Processing* is very capable of representing different types of representations, be it graphs, drawings, sketches or any other visual data type. Being that it is mostly a representation platform, additional libraries to represent the desired layouts will need to be used.

[5] https://processing.org/

3.6.3 JGraphT

JGraphT is a *Java* library dedicated to creation and manipulation of graphs. It started as a university project by Gaudenz Alder and has been kept updated, with the latest stable version being released in October 2020[6]. Along with the creation of graphs, this library contains several powerful algorithms that allow for the organisation and filtering of nodes in a representation. As the library is dedicated to the logic part of a graph, representation must be handled through a different method, *JGraphX* or *Processing*. *JGraphX* supports the representation of different types of graphs in the Swing platform and is limited mostly by the platform itself. It supports interaction with displayed graphs as well as manipulation of the current representation. Figure 3.14 is a representation of a graph modeled in JGraphT and represented with JGraphX.

Figure 3.14: JGraphT and JGraphX example.

3.6.4 Java Universal Network/Graph Framework

Java Universal Network/Graph Framework (JUNG) was created in 2003 and is maintained by Joshua O'Madadhain with the latest version being released on September 2016[7]. *JUNG* is an architecture developed with the goal of graph representation and is capable of handling most graphs types such as directed and undirected graphs, graphs with parallel edges and hypergraphs.

Additionally, the framework supports multiple algorithms from graph theory, social network analysis and data mining, features necessary for a good implementation of a family tree and UML diagram.

One of the many algorithms included in JUNG is VoltageClusterer, which clusters nodes based on their ranks (Fig. 3.15). This type of algorithm is a good example of how a good organisation is necessary for apt representation of information, and the inclusion of it and other different types of algorithms makes *JUNG* a powerful framework for managing graph data.

Its visualisation framework is capable of representing graphs defined within its architecture with different preset layouts and algorithms,as well as allowing the creation of custom ones. Being an open-source library, having good implementations, a diverse library of different algorithms and layouts and complete documentation made JUNG an potential candidate for representation of the diagrams for this project [O'madadhain et al., 2005].

[6] https://jgrapht.org/
[7] http://jung.sourceforge.net/

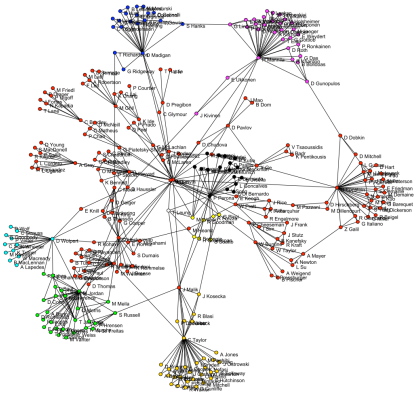
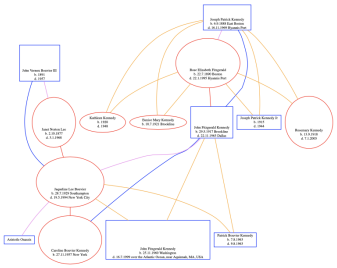


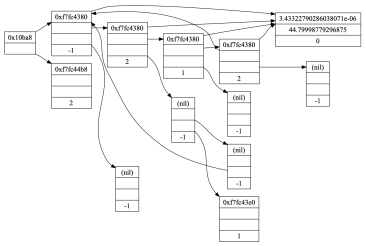
Figure 3.15: JUNG Graph using VoltageClusterer by CiteSeer.

3.6.5 Graphviz based applications

Graphviz is a platform dedicated to translating graphs in DOT language into graphics through the usage of different utilities. It supports several layouts which enable the creation of different types of graphs. Since it is open-source, it also allows for the creation of new layouts if needed. Figure 3.16 shows two representation types that could be useful in representing the diagrams for this work [Ellson et al., 2002].



(a) Family Tree graph.



(b) UML Class diagram.

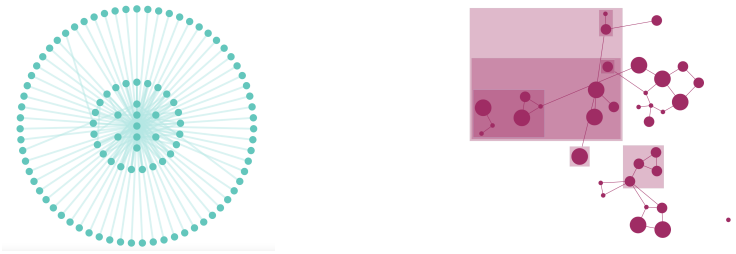
Figure 3.16: UML Class diagram.

Although *Graphviz* is very capable and provides easy methods for graph representation, it is mostly focused on static graphs and has little to no interactivity, especially when compared to previously discussed libraries, making it a harder to implement solution for the purpose of this work.

3.6.6 Cytoscape.js

Cytoscape.js is a *JavaScript* implementation of a graph library specialised in the modelling and visualisation of large amounts of data. The library is composed of a graph theory model and a graph renderer, making it an ideal solution of a full implementation for modelling and representation, unlike options like *Processing* and *D3.js* [Franz et al., 2016].

In addition to an extensive amount of supported layouts with different levels of organisation (Fig. 3.17) , the library is equipped with good documentation and examples which help a possible implementation.



(a) Concentric Layout.

(b) CoSE Bilkent layout.

Figure 3.17: Cytoscape layout examples.

Cytoscape.js also has a big focus on interactivity, with many features required for this work such as zoom, node selection and panning being already implemented on the library.

3.6.7 Comparison

All the previously referred to libraries and software are able to produce the graphs necessary for the implementation of this work and as such, in order to pick the most apt one, they where compared and the results are in Table 3.1.

Table 3.1: Summary of the tested representation libraries

Name	Documentation	Active	Type	Platform	Interactivity
Processing	Good	Yes	Visual	Java	Yes
JGraphT	Good	Yes	Model	Java	N/A
JGraphX	Good	Yes	Visual	Java	Yes
JUNG	Lacking (Old)	No	Model and Visual	Java	Yes
Graphviz	Good	Ending	Visual	N/A	No
P5.js	Good (New)	Yes	Visual	JavaScript	Yes
D3.js	Very Good	Yes	Visual	JavaScript	Yes
Cytoscape.js	Very Good	Yes	Model and Visual	JavaScript	Yes

They have different types of implementations, with *D3.js*, *Processing's P5.js* and *Cytoscape.js* opting for a *JavaScript* approach, giving the possibility of a web-based approach. This brings many advantages such as being web hostable and compatible with most devices, since it relies on the usage of a web browser, as well as having minimal overhead. They are also the three most recently updated libraries of this comparison, further solidifying the relevance of web-based approaches and the possibility of web-based platform being used. Additionally, web-based implementations can aid in remote testing if in person testing is not possible due to restrictions caused

by the ongoing pandemic. *D3.js* and *Processing's P5.js* are focused solely on representation and require the usage of another framework to handle graph modelling, like *JUNG*, *JGraphT* or *Cytoscape.js*.

Comparatively, *Processing*, *JGraphT* and *JUNG* use *Java* as their main platform, with *JUNG* and *JGraphT* having more emphasis in their graph handling capabilities while also having representation capabilities too. Their frameworks have features for graph manipulation and support several algorithms to aid in graphical representation and in highlighting of important parts of a graph.

Cytoscape.js and *JUNG* offer powerful architecture for graph modelling, visualisation and interaction, making them the most complete implementations for this work, as they alone can produce the desired results.

It is also possible to create a complete solution using a combination of other previously alluded to libraries, a mix of architecture and representation libraries. However, this method would require more implementation, with the added risk of features being incompatible. As such, using a complete solution like *Cytoscape.js* and *JUNG* should was prioritised.

Additionally, having the programming language of the selected libraries match existing file parsers for the data available for the graphs is ideal, although not strictly necessary. As stated previously, the file types for family trees and UML class diagrams tend to favour *Java* and *JavaScript* as their languages, further solidifying those two languages as the main candidates over Python.

Considering all these factors, *Cytoscape.js* was chosen as the primary representation library for this work.

3.7 Main platform architecture

Having considered the input devices and available technologies for both representation of graphs and importing of files, the final decision was to create a platform based in *JavaScript*. The architecture of the platform is divided into four main components - database, server, application programming interface (API) and its clients - all of them written in *JavaScript*. This allows the deployment of the application to be less constrained and the clients to be web-based, further increasing compatibility with available devices (Fig. 3.18).

3.7.1 Database

The database was created using *PostgreSQL* and its main purpose is to enable the use of accounts with secure and encrypted passwords so that users can store their graphs as well as keeping information stored (Fig. 3.19). As the database is simple, most database systems could to be used without any implications to the platform, resulting in the choice of *PostgreSQL* being a personal one. The storing of graphs is further complemented by an auto-saving feature, especially important in the event of an user closing the browser, the application crashing or any other event that would lead to data loss.

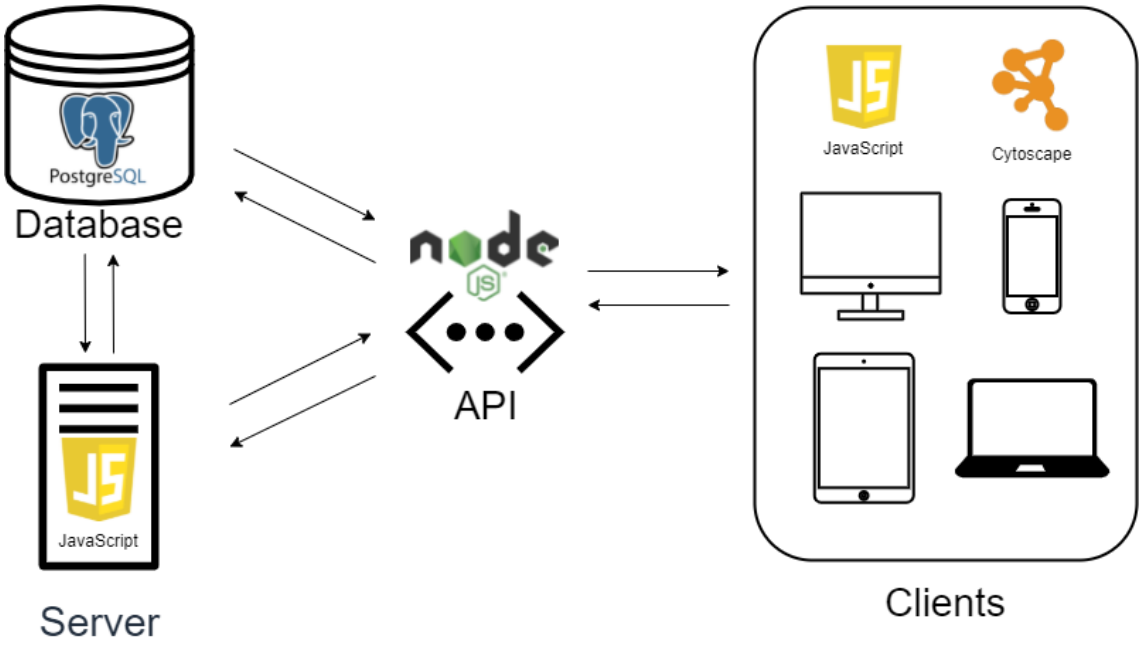


Figure 3.18: System diagram.

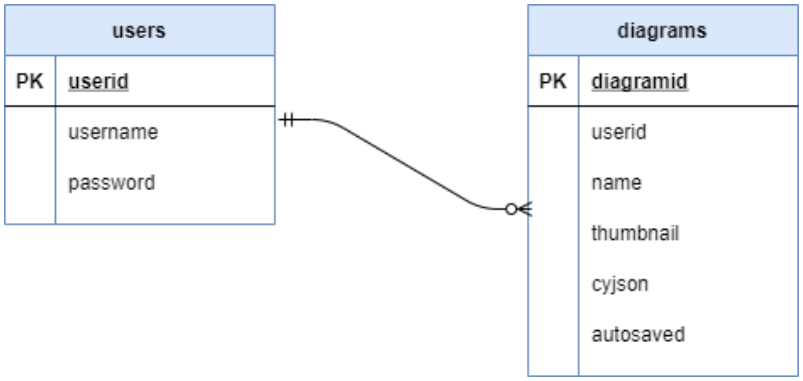


Figure 3.19: Database diagram.

3.7.2 Server and API

To ensure that the application remains light on the client devices, most of the processing is done on a server that is controlled via an API. When a client requests a given type of file to be opened, parsed and converted, it is up to the API to order the server to do all the processing and send the information ready to be laid out on a graph to the client. Furthermore, the API and server are equipped to handle the following:

- **Account creation and login** — To keep graph data safe while a user is working and to enable him to start a graph in a device and migrate to a different one, the API allows the creation of accounts on the database. When creating the account, the user is prompted to use a secure password, following a subset of rules. With an account created, it is possible to save graphs to the database and access them on every device where the account is logged in.

- **File uploading and temporary storage** — Many graphs have already been made in different platforms, and as such, files containing their information already exists. To use this information, the implemented platform must support the file format for the given graph types. The API enables the upload of *GEDCOM* and *XMI* files to the server, that are subsequently parsed and converted into working graphs if the file and contained data are valid. The original file is not kept in memory as it is not needed after parsing and converting.

- **File processing and graph information retrieval** — Enabling the use of data stored in files requires the conversion of that data into a format capable of being used by the main representation library *Cytoscape*. This process is done via a two-step method in which the data is converted into a JSON object and then into a data type usable by *Cytoscape*. By doing these steps in the server, the client devices don't require as much power to operate, allowing less powerful devices, such as phones and tablets, to use the platform.

- **Save and loading of diagrams** — To avoid issues with data loss, a save and load system was implemented. This allows for registered users to name their diagrams and save them to the database. This system also allows for an auto-saving feature to be enabled, further reducing the chance of data loss in the event of a system failure or user error. In case a user wants to remove their information from the CleanGraph platform, the saved graphs can be removed. Original *GEDCOM* and *XMI* files are only temporary during processing and don't require deletion by the user.

3.7.3 Client

For the purpose of this work, the client is intended to be how a user is able to interact with the platform. Considering the client is web-based, it is theoretically compatible with most devices ranging from mobile phones and tablets, to desktops and laptops. The client is responsible for the full control of the program, as well as what features are used at a given time. Most of the more resource-intensive features are triggered by a request to the API, which returns the information for the graph requested, or any other different query, whilst using the graph representation features, editing and exploration is done completely client-side. It is also the client that applies the final step of the graph representation, the layout. Since all the information required to build the graph is given by the API, the forming of the layout can be done fully client-side, avoiding synchronisation errors between the database and the client and to lightening the load on the API.

Chapter 4

Implementation

Having studied other implementations, devices and technologies to be used, the next step was the creation of the main platform *CleanGraph*.

4.1 Parsers

To guarantee that the solution is compatible with existing archived data, it requires the ability to read and translate data files for each category of graph. As alluded to previously, both graph types have many different types of files in which their information is stored, most of them proprietary, with a few exceptions. Genealogy software usually relies on GEDCOM as its main file of choice for importing data, as it has been supported for long and has proven its usefulness with this type of data. UML class diagrams on the other hand have many different proprietary file types due to the fact that the main file type for storing UML models, *XMI*, lacks some features that the more advanced editors use. However, it is the standard file type and is widely adopted by most UML class diagram editors.

For the proposed solution to use the chosen file types, their information needs to be converted to be compatible with the representation style. In both cases, family trees and UML class diagrams, this conversion is made in two steps.

By converting the files initially to a JSON object, the data can be better controlled, organised and filtered to eliminate possible errors and possibly extract extra data from corrupted files. This two-step approach also gives the possibility to create and use functions agnostic to the graph type being converted.

4.1.1 GEDCOM conversion

Initially, the support for GEDCOM X was considered, but after testing available parsers and gathering data for testing, it was discarded as there wasn't enough support for the data type and the available data was negligible when compared to GEDCOM. These factors and the continued support for the GEDCOM made it the file format of choice. As stated previously, the conversion from file to graph will be done in two major steps, parsing the file to a JSON object containing all

the relevant information for the final graph, and a subsequent conversion from a JSON object to elements that are accepted by *Cytoscape.js* and that allow for the features needed. The first step was helped by the usage of a GEDCOM parser named *gedcom.json*. This parser allows for the conversion of the text based GEDCOM file into a more manageable data format (Fig. 4.1).

Figure 4.1: Example of GEDCOM to JSON object of a person in a family tree.

The data provided by the parser is separated into two groups:

- **Individuals** — Comprised of all the people in the family tree with all the information pertaining to them, such as names, gender, important dates, details about events in the person's life and what family they belongs to.
- **Relations** — Contains all of the families in the family tree. In GEDCOM, a family is made of two parents and their children. The relations also contain information about events such as marriage or divorce with accompanying dates, and the individuals that belong to the family.

After being parsed, the information can be converted into a format that is capable of being represented in *Cytoscape.js*. To achieve this, a converter was created, capable of translating the parser information into usable data for *Cytoscape.js*. The converter first takes the information from each person node stored in the *individuals* and adds them to the *Cytoscape.js* instance. In the nodes, all information pertaining to the node is stored: name, sex, birth date, death date, gender, image and notes. The converter takes special steps when creating dates, converting them from raw data to a text based form that allows for a better representation of time. To preserve some of the data, even if it is invalid or non-standard, the converter will attempt to fix invalid names and dates. If it fails to do so, that information will be ignored to avoid issues. In a second phase, converter takes the parsed relation information and creates all the links between the previously created nodes, differentiating between types of relationships such as marriages and divorces and families with children. When creating these links, information related to the corresponding event, such as date and location, are stored to be used later. With all the information on the graph, a layout algorithm can place the nodes and edges in the appropriate position.

4.1.2 XMI conversion

Unlike GEDCOM, there are few *XMI* UML parsers, with most of them being outdated and not compatible with the newest version of *XMI* 2.1, or being proprietary and unable to be used for the purpose of this work. Considering that supporting *XMI* 2.1 ensures support for the older versions, a parser was developed, capable of parsing most of the *XMI* 2.1 standard into a JSON object. The

parser was implemented using *Simple API for XML*, also known as *SAX*, in its *JavaScript* version, *SAX-js*.

SAX-js was chosen for this work for its simplicity of usage and compatibility with the *XMI* file format, since *XMI* is derived from XML file format. The API works by parsing through an entire file, recording the tags from each XML element and subsequently providing events triggered by each of the found tags. For the purpose of parsing an *XMI* UML document and creating a UML class diagram, the required tags are parsed as follows:

- **Class / Association Class** — Contains information related to a class including its type and name. Upon finding this tag, the parser creates an object in which to store all the parts relating to the class, such as properties, operations and stereotypes.
- **Operation** — Upon encountering the operation tag, an operation is added to the corresponding class tagged in its attributes. Operations have properties such as name and visibility, and support the addition of parameters such as function returns and inputs and outputs.
- **Parameter** — Parameter tags are used to fill the parameter data in operations. They contain properties such as name and type of parameter, as well as what kind of parameter it is, return or input-output.
- **Association** — In the *XMI* format, the association tag is used for the three different types of UML relations: associations, compositions and aggregations. On this tag, an association element is created with the members of the association. Other properties are added in different tags.
- **Literal String** — If the parent tag of a literal string tag is an association, the tag houses all the information pertaining to an association's multiplicity upper and lower value. If the parent is an attribute, then the literal string contains information about the default values of the parent attribute.
- **Property** — The property tag is used for many different types of information depending on the parent tag it has. If the parent tag is of the type association, then the property tag carries information about an association role, type and navigability of one of the members of the association. If the parent is a UML class, then the property tag refers to an attribute in the class, and the name, type and visibility are stored.
- **Dependency** — The tag contain information about the supplier class and the client class of a dependency.
- **Generalisation** — Like dependencies, a generalisation tag contains the information about the parent and child of a generalisation.
- **Data Type** — The *XMI* file format categorises data types such as integers, strings or other custom variables as special *dataTypes*. These *dataTypes* need to be stored in order to be

referenced in the UML class diagram later as they contain the ID to be referenced and the name of the corresponding variable.

- **Pre-defined constraint** — Pre-defined constraints are defined by the constraint tag. These constraints are made between two associations and have a name that defines their meaning.
- **User-defined constraint** — User-defined constraints are defined by the opaque expression tag. These constraints are usually represented in UML class diagrams in a similar way to comments, and contain a name, a text body and a class reference.
- **Comment** — Comment tag contains the name and text body of a comment for an associated parent tag.

The resulting output from the parser is formatted as:

```
model = {
    types: [], // Classes
    associations: [], // Associations between classes
    dependencies: [], // Dependencies between classes
    generalizations: [], // Generalizations between classes
    comments: [], // Comments for classes
    userConstraint: [], // Constraints of classes
    predefinedConstraint: [], // Constraints between associations
    dataType: [] // Data type references
};
```

Figure 4.2: *CleanGraph XMI* parser output.

With all the information stored in a JSON object, it is then passed along to a converter, similar to the process for *GEDCOM* processing. To transform the data into a format compatible with *Cytoscape.js*, the model obtained via the parser is iterated. As with *GEDCOM*, the converter creates the UML classes and places them in the diagram, and subsequently creates all the connections between them. Unlike in the family tree module where each node contains one person, the nodes in the UML class diagram module are composed of three different sub-nodes containing name, attributes and operations of an UML class, all included in a parent node that represents the main class. After all the nodes and connections are made, a layout algorithm can be ran to ensure the UML classes are positioned appropriately.

4.2 User Interface

A common practice between most genealogy programs and UML class diagram editors is the use of additional windows, popups or auxiliary bars to aid navigation and to give information about specific parts of a graph. To keep the user interface of *CleanGraph* easy to read and still provide information that the user can use, different types of bars where created (Fig. 4.3):

- **Sidebar** — A bar that is always present on the left side of the *CleanGraph* user interface, UI, and that retracts when not in use. The main goal of this UI element is to house shortcuts,

allowing navigation between most of the features of the program such as opening the family tree module, UML class diagram module or the user account and saved data.

- **Auxbar** — The main goal with the addition of an auxbar to the right of the screen is to allow for more complex data to be shown to the user. This ranges from information not shown in the graph due to space restriction to more complete versions of the data that would clutter the graph representation if displayed all the time. Additionally, this auxbar is hidden if the screen size is not appropriate, as in small browser window or mobile phone in portrait mode.

- **Topbar** — The smaller of the three bars, it allows a user to login or register, and when the user is logged in, also allows for naming of the graph, saving and exporting, as well as enabling the auto-save feature.

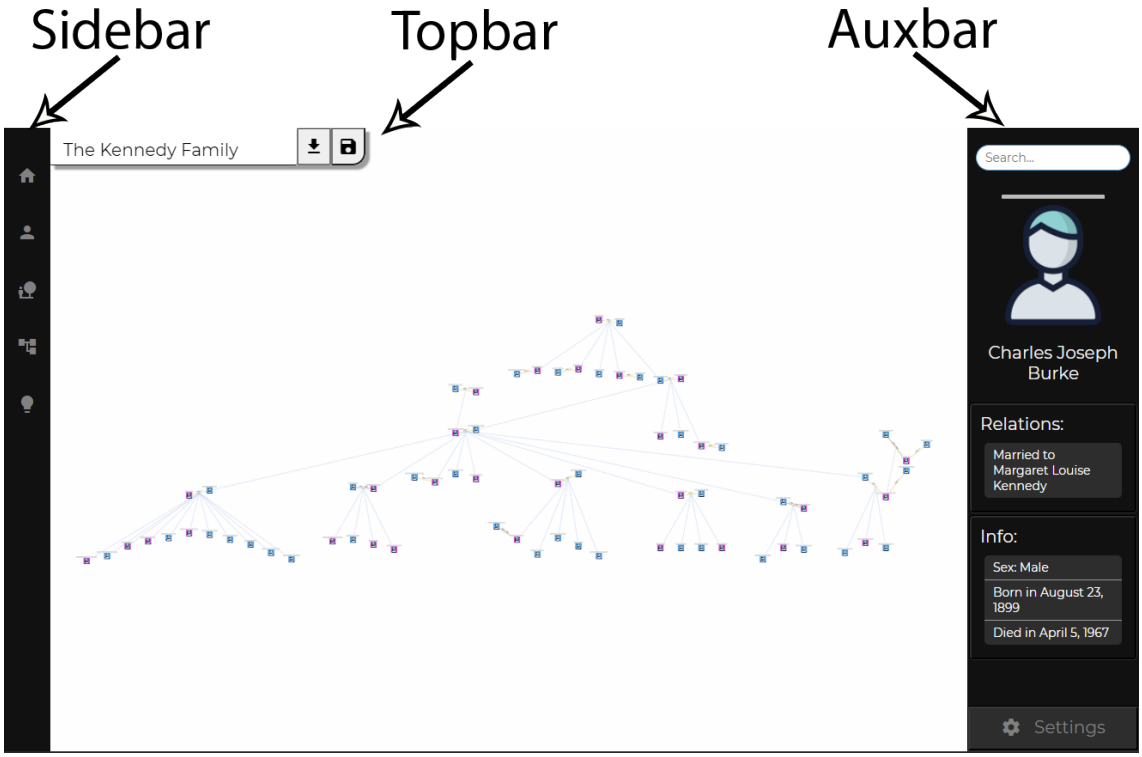
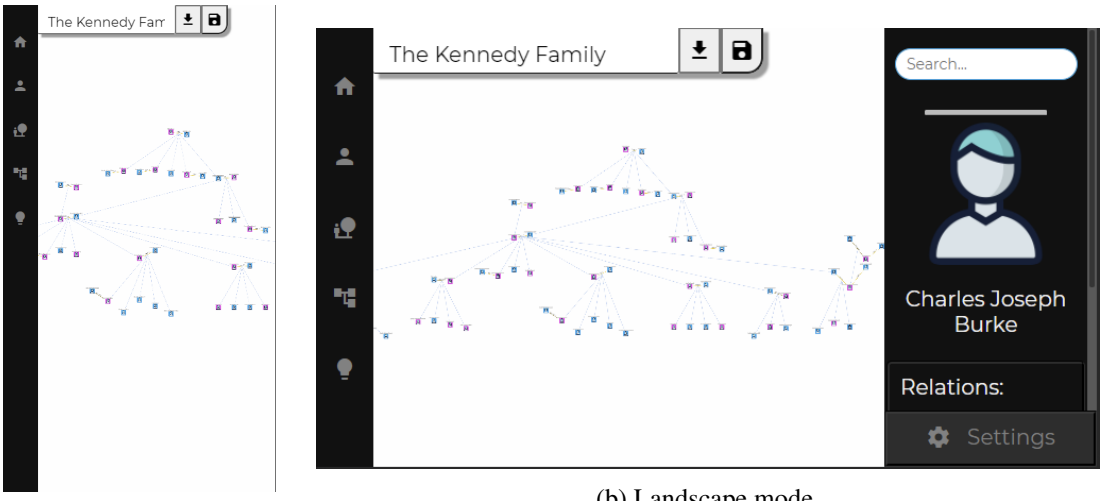


Figure 4.3: CleanGraph UI.

CleanGraph's UI is responsive, and as such, adapts to big computer screens as well as smaller tablet and smartphone displays. Allowing this adaptation further solidifies the multi-platform approach taken when developing the platform. For tablet devices, the UI remains similar to desktops. However, on mobile phones, some adaptations were made to fit the considerably smaller screen. In portrait mode, the auxbar bar is hidden to give as broad of a representation as possible (Fig. 4.4a). If a user desires to see more specifications, rotating the device to landscape mode will bring the auxbar, which the user can scroll through (Fig. 4.4b).



(a) Portrait mode.

(b) Landscape mode.

Figure 4.4: *CleanGraph* in a mobile phone.

The UI was designed with cursor and touchscreen controls in mind, with the occasional usage of the keyboard for text input. Hovered elements in the Auxbar will highlight corresponding elements on the graph, and clicking them will select and pan to it. Additionally, hovering nodes or relations will provide the user with extra information without complicating the graph representation (Fig. 4.5).

(a) Hovered node.

(b) Hovered edge.

Figure 4.5: Hovering in *CleanGraph* family tree graph.

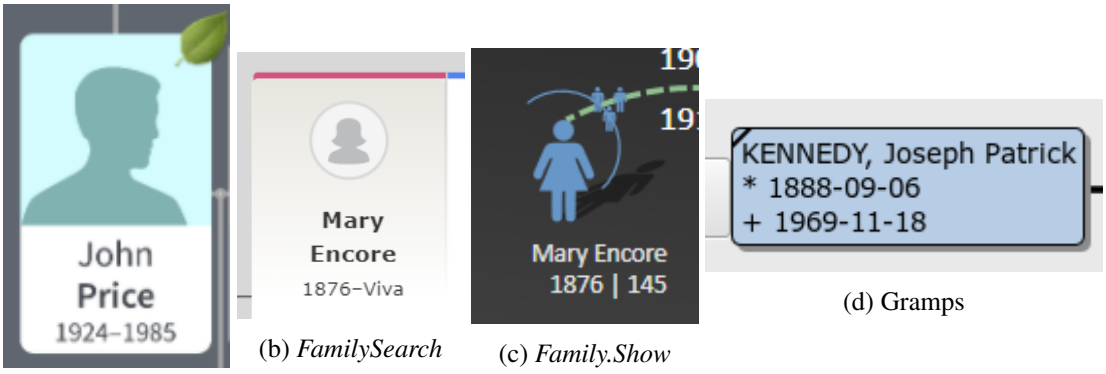
Although these examples are for family tree representations, the fundamental behaviour of hovering and clicking is the same for UML class diagrams, with minor differences due to the type of data displayed.

4.3 Family Tree

Creating an appropriate family tree requires the constructed graph to have aptly chosen features that can help guide users when reading the information contained in the graph. To guarantee this, it is important that the representation of crucial aspects of the family tree, such as people nodes, their relations (marriage, divorce and children) and hierarchy is well implemented.

4.3.1 Choice for Nodes

One of the most important features of a family tree is the representation of the people in it. Given that the proposed solution uses a graph approach, using nodes and edges to represent people and their relations, it is important to keep the people nodes simple but still containing all the important information. Genealogy platforms alluded to previously, *Ancestry*, *FamilySearch*, *Family.Show* and *Gramps*, have different representations of nodes, but it is possible to extract some similarities between the information represented in them such as name, image, birth and death dates and gender (Figs. 4.6).



(a) *Ancestry* (b) *FamilySearch* (c) *Family.Show* (d) Gramps

Figure 4.6: Node representation in different programs.

It is useful that this information is readily available at a glance, but there is still plenty more information about an individual than just this, and although it does not need to be always on display, there must be ways to see it if the user desires to. The already existing platforms chose from different types of display, mostly consisting of sidebars, popup menus or even altering the nodes to allow for further information to be displayed.

With this information in mind, the following node format (Fig. 4.7) was created:

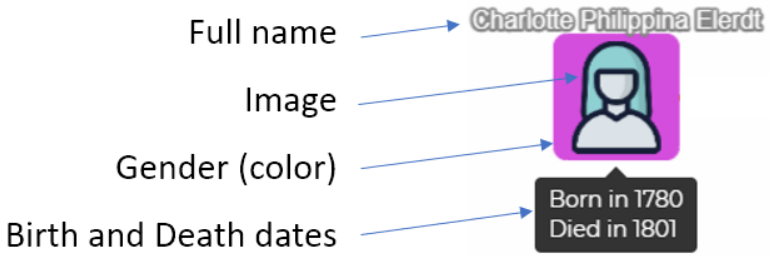


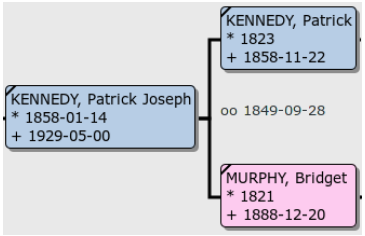
Figure 4.7: *CleanGraph* node.

At a glance, the current gender, first name and surname can be seen and if the user chooses to hover the node, the information about birth and death is shown as a label. If the person has a photo image, the node will have a highlighting border with the colour of the gender instead.

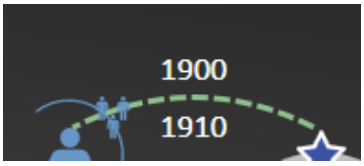
Additional information can be found by clicking on the node. This opens an auxiliary bar that contains details about that person, such as notes and more complete dates. Furthermore, the auxiliary bar allows the user to navigate a tree by selecting a person's relation, such as marriage, or its descendants or ancestors.

4.3.2 Choice for Family Links

Links between nodes convey information of how different people in a family tree relate to each other. In most genealogy programs these links are very simple and serve to connect people based on marital status, descendancy or ancestry. Since some of these relations can be dated, such as marriage/divorce dates, some programs like *Gramps* (Fig. 4.8a) and *Family.Show* (Fig. 4.8b) chose to pass that information as a label on the link.



(a) *Gramps*



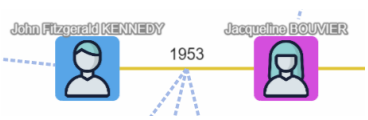
(b) *Family.Show*

Figure 4.8: Link representation in different programs.

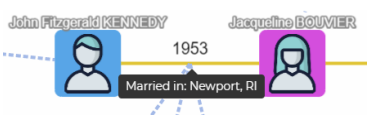
Although these connections can be used to convey information to the user, it doesn't mean they should always be used to do so. Overloading them with information can result in a cluttered graph design.

To avoid this, links representing relationships can have different symbols, shapes, labels, form and colours, allowing for the distinction between the different types. Additional information can also be added through labels on the link or on the ends of the link (eg. in a marriage, indicate which node is which, Husband-Wife, Husband-Husband, Wife-Wife) or hovering, which can contain information about the location of the event.

When creating the link representation in *Cytoscape.js* different approaches were tested, resulting in a final version that is similar to what most genealogy programs use, with some improvements. If a relation is a marriage or a divorce and has extra information such as date and location, instead of hiding that information in sub-menus like *FamilySearch* and many other programs, it can be simplified and shown in the link (Figs. 4.9).



(a) Link representation



(b) Link when hovered

(c) Triple link

Figure 4.9: Link representation

Using colour helps create a simple separation between both types of relations, marriage and divorce, and it can be further enhanced with icons if needed. To better use the space in a graph, a representation of the marriage/divorce date is done via a label on the relation. This date is simplified to show only the years of start and end of a relation, if the information is present. Through the use of hovering, additional information about the location of the marriage/divorce can be visualised. Further information about the relation is available in the Auxbar.

4.3.3 Layouts

CleanGraph supports two modes of auto-layout, a more traditional hierarchical mode called normal, and a layout that uses the vertical axis of the graph to represent the date of birth of the people.

The first layout starts by locating the oldest person in the family tree by filtering all possible candidates and finding the one with the earliest birth date. If there are no birth dates available, the layout algorithm will pick a person who has no parents available in the tree. This was chosen as a backup method to help the tree building algorithm minimise the possibility of overlapping people and relations. The first person in a tree is always placed on the X:0 and Y:0 coordinates in the graph due to it not having any other nodes to position in relation to. After being placed, that person is added to the *visited* list to avoid it being moved unintentionally.

The selection and placement of the remaining people is dependent on the last placed person and is divided into three different sub-steps:

- **Place Partners** — Firstly, the layout algorithm will pick all the partners from the last person placed. If there are none, the step is skipped. If its only one partner it will attempt to place it to its right. If it has more than one partner, it will attempt to place them to the right or to the left of him, whichever is closest.

- **Place Parents** — After placing the partners, the algorithm will attempt to place the last person's parents on top of him. If there are no parents, the step is skipped.

- **Place Children** — The final phase after placing a person is placing its children. For the tree to be spread whilst avoiding collision and overlapping of sub-families, the space to be occupied by the grandchildren is previously calculated. This way, when placing the children, the space available already accounts for the space the other generations will take. Additionally, before the placement of the children, the order in which they are positioned is defined. To avoid potential collisions with already placed people, the children are ordered by partner number. If a children node has more partners, it will be placed farther away from the parent. In the event that there are no children, this branch of the family tree is terminated.

To avoid overlapping when building the family tree, the layout algorithm will first verify if there is the possibility of collision with already existing nodes on the tree. If there is no risk, the node will be placed in the specified position. However, if a possible collision is detected, the node will not be placed. Instead, the function will act in one of the two different modes available: move

or push. If set to *move*, the node will be moved forward or backwards in the X axis (horizontal axis) to avoid collision whilst trying to keep it as close to the original placement position. If set to *push*, the node will be placed on the defined place and will push the other nodes that are already placed on the tree forward. This will happen recursively, one node being pushed will result in every node in front of it (in the horizontal axis) to be pushed as well. After everything is placed, the tree is presented to the user, like in with the *Kennedy Family* example (Fig. 4.10).

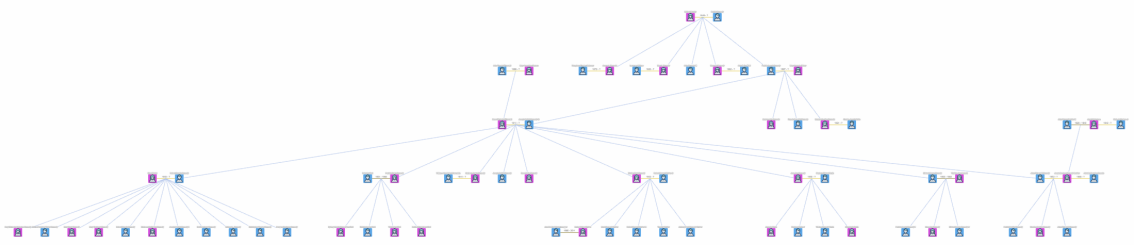


Figure 4.10: Full Kennedy tree on *CleanGraph* with normal layout.

The second layout type uses the vertical axis as a way to represent the birth year of the people in a family tree, and starts by running the first layout. This is done to facilitate the positioning of people in the horizontal axis. As a first step, the algorithm looks for the oldest family member and the youngest to establish the range of years to be fitted in the graph. The age range is then multiplied by a user-defined value that will regulate how far or close together the people are vertically. If the family data imported to *CleanGraph* has every family member with appropriate birth dates, all the nodes are repositioned to the correct vertical position according to their age. However, even if there is no birth date for some of the people in the tree, they still need to be placed appropriately to create a coherent visual representation. There are several different cases that require special handling due to missing information:

- **One individual in a relation has no birth date** — If a member of a relation has no birth date but its partner does, they are both put on the same level vertically. This levelling occurs irregardless of the number of partners in a relationship. For example, if in a four person relation there are three members without birth dates, they are all levelled to the same vertical position as the one member with the date available.

- **One individual has no birth date but its parents do** — In the event that a person has no birth date information available, but one of its parents has, it is placed the equivalent of 30 years below the parents. This value was chosen due to it being the average mean value of years between generations. Additionally this value is modified by the same height multiplier defined previously. This method is also applied to people in a relation where none of the members have birth dates, with the key difference being that the search for parents with available birth dates is done across all members.

- **One individual has no birth date but its children do** — As with the previous event, if a person has no birth date but at least one of its children does, the person is positioned

the equivalent of 30 years above them. This is also applied if all people in a relation have missing birth dates.

- **One or more individuals have no birth dates but their siblings do** — If one or more people in a family have no birth dates, but their siblings do, their height is calculated based on the average date of birth of their siblings. This is preferable to the 30 year method as it can be a closer approximate to the real date position.

With the vertical position now defined, the user is presented with the representation like the *Kennedy Family* example (Fig. 4.11) where it is possible to visualise the age difference across the whole family.

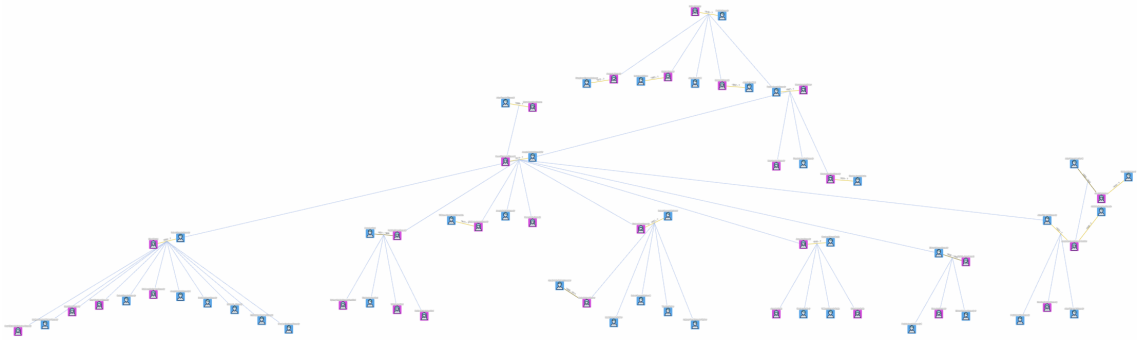
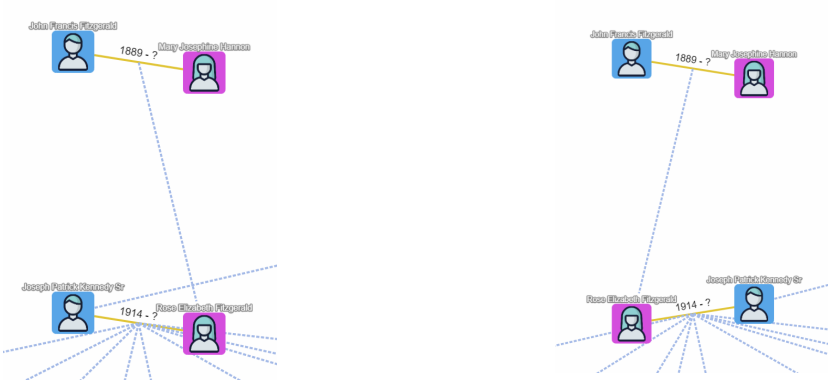


Figure 4.11: Full Kennedy tree on *CleanGraph* with age layout.

Due to the way the algorithm chooses to place the nodes, there are some cases where some relations might end up crossed. To avoid this, after the nodes are placed, the couples with relationships are revisited and checked for crossed relations. There are two main instances where this method produces advantageous results: when people in a relation have their parents in opposite sides (Fig. 4.12) making the parent-child relation become crossed, or when no crossing occurs but the positioning is sub-optimal (Fig. 4.13).



(a) Crossed (b) Solved

Figure 4.12: People with crossed relations.

(a) Opposite

(b) Optimal

Figure 4.13: People with sub-optimal positioning.

4.3.4 Auxiliary bar - Auxbar

In the family tree module of *CleanGraph* the auxbar takes the role of housing the search bar (Fig. 4.15, as well as the displaying of relevant information to what the user is doing at a given time.

When a user starts writing text into the search bar field, all the people in the tree whose name matches the search query get highlighted. As the user writes, the highlighting matches the corresponding people (Fig. 4.14) and an additional list underneath the search bar appears.

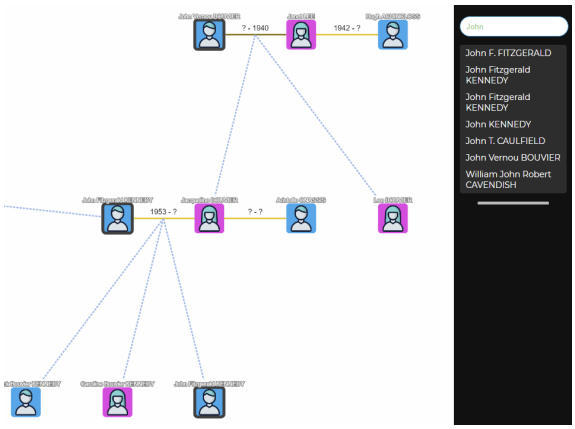


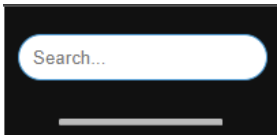
Figure 4.14: CleanGraph people highlight

This list contains all the matching people (Fig. 4.15b) in the tree (up to 100 names to avoid performance problems), and hovering the elements on the list highlights the desired person. If clicked, the graph will rapidly zoom in the corresponding person.

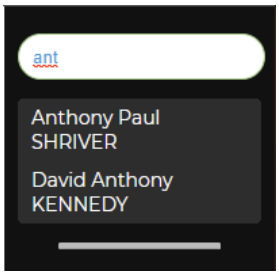
Selecting a person by clicking on them or searching for them in the search bar will turn the auxbar into a representation of the information available about that individual (Fig. 4.16).

In this representation there are six major components:

- **Image** — Traditionally the file format *GEDCOM* does not carry information about images, and as such in most cases, when importing *GEDCOM* files, the image will be that of a male, female or unspecified type, depending on the gender information available. If the person is edited, the image can be added and stored in the platform.



(a) Closed Search



(b) Open Search

Figure 4.15: Search bar in CleanGraph

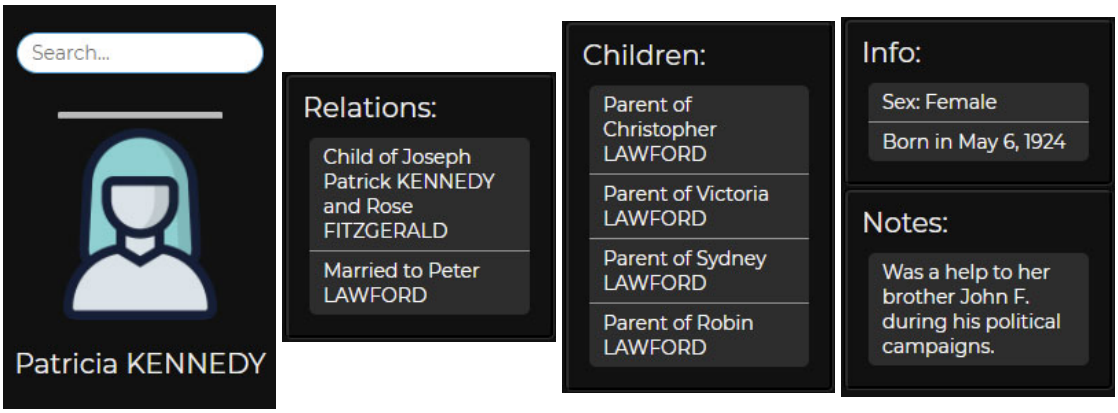


Figure 4.16: Auxbar when a person is selected

- **Name** — The name that is shown is the *Fullname*, as defined by the *GEDCOM* standard. *Fullname* is a combination of the first and last name of the person.

- **Relations** — The relations tab serves multiple purposes in the sidebar. It allows for a text based representation of direct relations between the selected person and other people, be it in marriage/divorce or its parents. Additionally, hovering the elements on the list will highlight them on the graph, allowing for easier interpretation of the data. Hovering the parents of the selected person will highlight the corresponding pair of parents in the graph, and clicking the element will pan and zoom to the parents. Likewise, hovering a marriage/divorce in the relations tab will highlight both the selected node and the marriage/divorce partner, and clicking will pan and zoom to the relation.

- **Children** — In a similar way to how relations are treated in the relation tab, the children tab houses all the descendants of the selected node. This includes children from multiple marriages if they have the selected parent in common. As with the relations, hovering the children on the list will highlight them, and clicking on the element on the list will pan and zoom to the child.

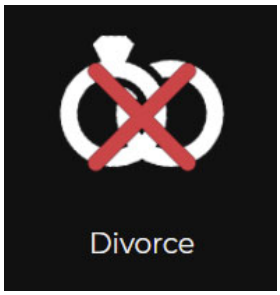
- **Information** — The main goal of the information tab is to contain all the relevant personal information of a selected person. The possible information is the gender, birth and death

dates, as well as birth and death places. Additionally, the dates are displayed with as much information as possible, accounting for special cases such as incomplete dates, rounded or approximate dates or estimated dates or even time periods.

- **Notes** — *GEDCOM* file type supports the addition of special types of information in the form of notes. These notes can contain many varying types of information such as parts of the life of a person, history about their health and many other events such as baptisms, bar mitzvah, etc.

To complement the information present when hovering a relation, the auxbar also adapts to show information when a relation between people is selected (Figs. 4.17).

(a) Marriage



(b) Divorce

(c) Information

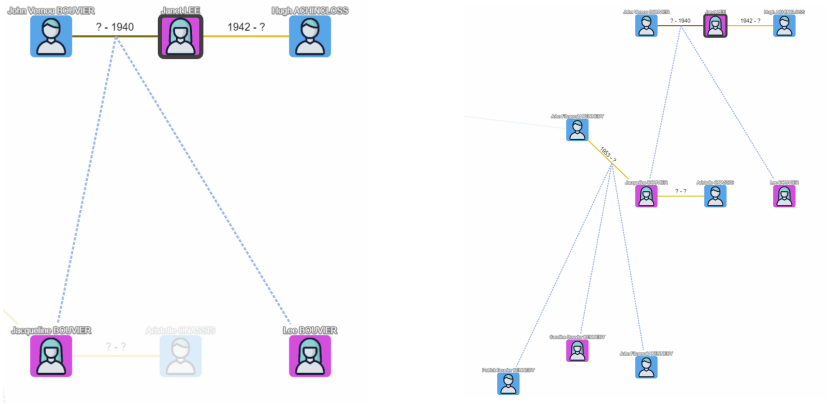
Figure 4.17: Selected relation in auxbar.

The main goal with the addition of this feature is to increase the amount of information present without hindering the visualisation of the graph. By displaying simplified version of the relation information in the link and by hovering, a user can scan the graph easily, but if he wishes to see more about that relation, he is able to select it and be informed by the auxbar. This extra information depends on the type of relation selected. For marriages, the starting date and marriage location is displayed, for divorces, both the original marriage date and location is shown, as well as the divorce date and location. Selecting relations such as the birth of a person is also possible, and information related to that event will be displayed.

4.3.5 Generational Highlight

Like most genealogy programs, *CleanGraph* allows for exploration of its graphs by panning and zooming. This traditional method is enough for the interpretation of data, but it can be complemented by other features such as *Generational Highlight*.

By double clicking/tapping on a person in the graph, the graph will pan and zoom to show all the nearest descendants, relations and ancestors of the selected individual. While the pan and zoom is happening, all non related people and relations are phased out to help the graph appear less complex (Fig. 4.18).



(a) One generation (b) Two generations

Figure 4.18: Generational Highlight

In the auxbar menu, a user can select the amount of generations to be highlighted through a slider, allowing for only the related information to be shown.

4.4 UML class diagram

For creating a representation method for UML class diagrams, special care is taken due to the diagram type's strict rules and the needed clarity of the representation.

4.4.1 Choice of UML Node

The representation of a UML class diagram table in a *Cytoscape.js* node poses some challenges based on how the framework works. Compound nodes and edges require special care to properly represent UML class diagrams following the standard set by OMG.

Unlike family trees containing different amounts and types of data per node, UML class diagrams feature more standardized nodes, always comprised of three parts: name, attributes, operations. The creation of this node in *Cytoscape.js* can be achieved with different methods:

- **Creating a node with a label comprised of all the data of the UML class table** — This approach is functional but is less visually attractive, less intuitive and does not allow for customisation of the node appearance.

- **Creating a node in which the background image is the table** — Doing this would require a processing layer that would translate the UML class table information into a compatible image file and applying it to the node. Although this could provide a good look for the

node, it adds processing overhead and can hurt interactivity with the node's parts, as the image would not be interactive, and editing would require the node image to be remade.

- **Creating a compound node comprised of three child nodes, one for each of the UML class table components** — The main issue with this node creation method is the inability of traditional *Cytoscape.js* layouts to be mindful of parent nodes of compound nodes, choosing to ignore them and positioning child nodes regardless of their parents. However, these issues can be offset and fixed using different methods. *Cytoscape.js* inbuilt layouts are not compound node aware, but custom created layouts can be, so creating a customised layout for UML class diagrams solves the issue. Additionally, layouts can be run on specific elements of the graph, and as such, specific adjustments to compound nodes can be made.

Both the attribute and operation nodes have details that require implementation. Attributes in the attribute node can be public, represented by a "+", private, represented by a "-", or protected, represented by "#". Operations in the operation node must denote if they are "in", "out" or "inout".

With these considerations in mind, the UML class node (Fig. 4.19) was created:

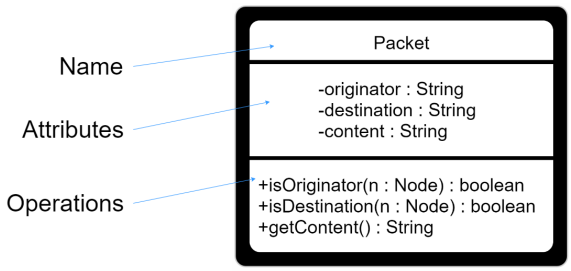


Figure 4.19: CleanGraph UML class node.

Like UML class representations in other programs, it is possible to see all the important information at a glance.

4.4.2 Choice of representations for UML relations

Although family trees allow for customisation in the links between the nodes, UML class diagrams have well-defined relationships that need to be represented as accurately as possible. By following the designs required by the UML class diagram specification, the following relations were created (Fig. 4.20):

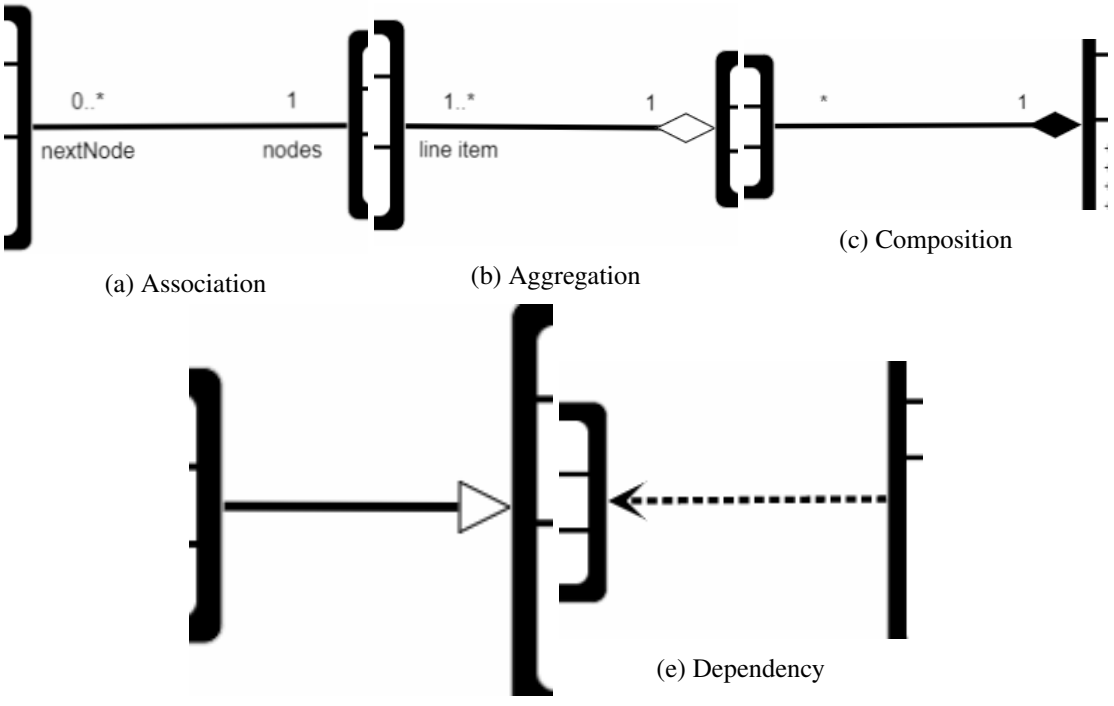
- **Association** — Associations in UML class diagrams are usually represented by a line between two classes or itself. These associations carry different amounts of information that must be represented to respect the UML class diagram standard specified by OMG. Apart from simple labels naming the association, associations can have multiplicity and roles. They are usually represented by a range of values and names near a class and can be present on both the beginning and the end of a association (Fig. 4.20a). Additionally, associations can have navigability directions that are represented by thin arrows in the ends of the relation.

- **Aggregation** — An association that is directed, representing that one class owns another class. This relation is represented by a hollow rhombus on the end of the relation that attaches to the owner in the relation. As with associations, compositions can also have defined multiplicity and roles on their owner or target (Fig. 4.20b).

- **Composition** — Relation similar to an aggregation that represents the same connection but makes the owned class dependant on the owner class. This difference compared to aggregations is represented by the presence of a filled rhombus, as opposed to a hollow one in aggregations (Fig. 4.20c).

- **Generalisation** — Generalisations, often called inheritances, are directional relations that have a parent class and a child class. The direction of the generalisation is represented by a singular hollow white arrow pointing from the child class to the parent class. The representation of a generalisation from many children to a parent can be done with multiple singular connections to the parent class or with all the connections of the children converging into a single line with just one hollow white arrow pointing to the parent (Fig. 4.20d).

- **Dependency** — Like generalisations, dependencies are directed relations with a supplier class and a client class. Traditionally they are represented by a dashed line with a thin arrow pointing in the direction of the supplier class (Fig. 4.20e).



(a) Association (b) Aggregation (c) Composition

(d) Generalisation (e) Dependency

Figure 4.20: UML relations in CleanGraph

In previous chapters, representation platforms where studied, and the chosen platform, *Cytoscape.js*, was tested and allowed for the representation of the needed relations with a small amount of tweaking to the source code.

4.4.3 Layouts

When generating a layout for an UML class diagram it is important to take in consideration the problems that can arise in the representation. The first overview layout that is presented to a user after importing an UML class diagram starts by searching for the UML class which contains the highest amount of relations to other classes. After selecting the class, the positions around the main class are calculated. For example, in the case of a class with four relations, the layout algorithm will try to place the related classes evenly spaced around the 360 degrees available around the class. The main function working towards the calculation of the class placement can be found in Appendix C.

After placing the first related classes the algorithm will run recursively to find position for all the placed classes. These classes will run the same placement algorithm as the first class but in a more restrictive angle, instead of the 360 degrees of the first class. The angle is calculated based on the previously placed classes and the available space between or near them.

4.4.4 Auxbar

The approach taken for the sidebar is similar to the one that was taken with the family tree module, with adaptations specific to the UML class diagram specification. Like in the family tree module, the search bar feature is located in the auxbar. In the UML class diagram module, the search function is adapted to search for more than just the name of a class. When a user starts typing in the search bar field, the classes with the name corresponding to the search query are highlighted. Additionally, classes with attributes, operations or variables in operations with corresponding names to the search query are highlighted (Fig. 4.21).

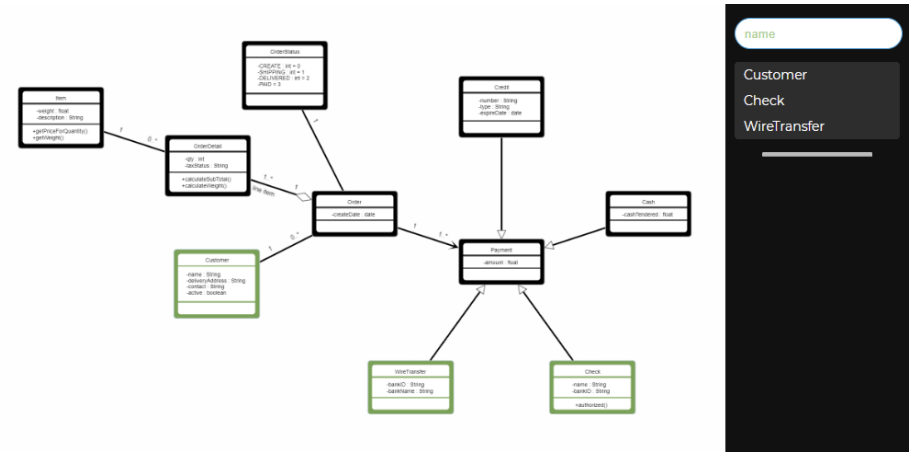


Figure 4.21: CleanGraph UML highlight

As the user types the search query, the results list is filled with the classes that contain the corresponding names, attributes and operations (Fig. 4.22b). Hovering each of the results will highlight the corresponding class, to aid the navigation through the diagram and clicking it will pan and zoom to it.

(a) Closed search

(b) Open search

Figure 4.22: Link representation

Selecting an UML class by clicking it on the graph or in the search bar will select the class and change the auxbar to represent the information available about the UML class (Fig. 4.23).

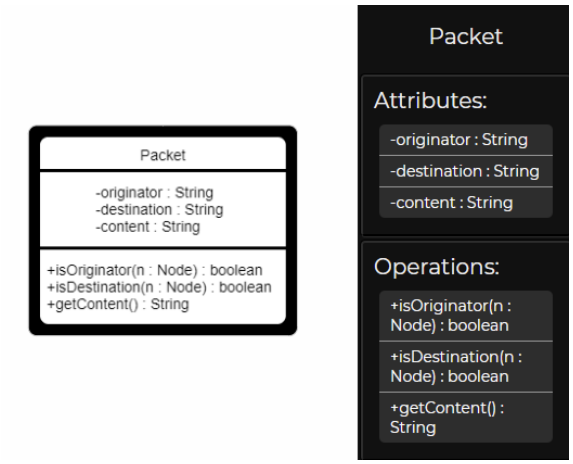


Figure 4.23: Auxbar when a class is selected

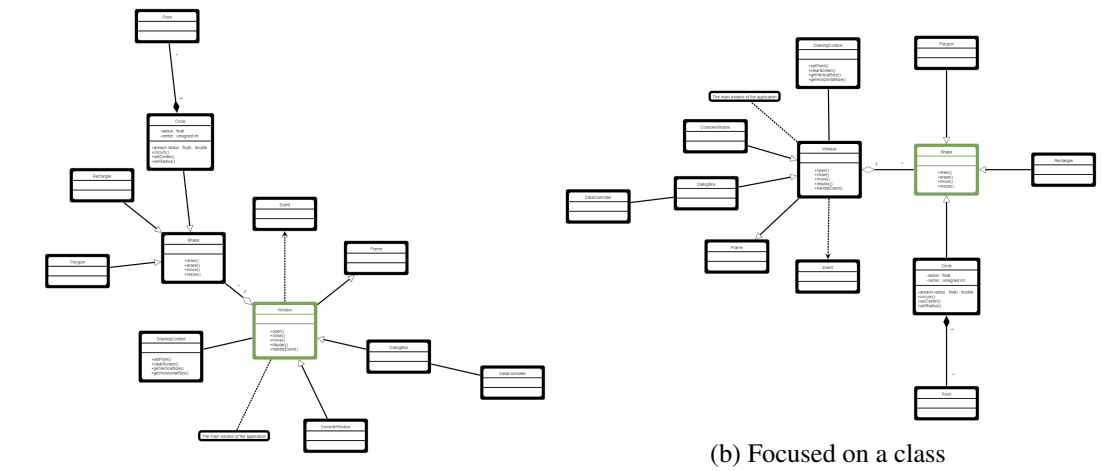
This information is divided into three different segments:

- **Name** — The name of the selected class
- **Attributes** — The attributes tab contains all the attributes of a given class, their visibility, their type and default values.
- **Operations** — Likewise, the operations tab contains all the operations of a given class, including their returns, inputs and outputs, visibility and default values.

4.4.5 Reorganise

To help highlight the relation between the different classes in an UML class diagram, *CleanGraph* allows for a user to center the layout around a selected class. By double-clicking/double-tapping

a UML class, all other UML classes reorganise to position the selected class as the center point of the graph, with classes that originate from it in its surrounding (Fig. 4.24b).



(a) Default

(b) Focused on a class

Figure 4.24: Appearance of a UML class diagram in *CleanGraph*

Chapter 5

Testing methodology and results

Testing *CleanGraph* is a required step in order to confirm that this platform addresses the issues that it is meant to fix, as well as to gather information on user experience and their opinion when using the platform. The testing is divided into two categories: graph quality assessment, where *CleanGraph*'s representations are compared to other platforms in how the information is displayed and how the difficult cases are handled, and user testing, where users from the genealogy community were assigned a set of tasks that allow for the comparison of performance between *CleanGraph* and other genealogy platforms. As the UML class diagram module of *CleanGraph* is not as feature-rich as its family tree counterpart, the user tests were made exclusively for the family tree module.

Even though the second module is not tested by the test subjects, some of the information and criticism gained from the family tree module testing can be applied to it. This is one of the advantages of having a common representation platform with an user interface that is shared across the modules.

5.1 Graph quality assessment

CleanGraph's graphs for both family trees and UML class diagrams provide an alternative representation to what other programs offer, aiming to enable better perception and interaction with the information.

In previous chapters, issues found in different platform's graph layouts where considered, as well as their methods for avoiding or fixing them, and will now be discussed in relation the developed solution.

5.1.1 Large data sets

As previously described, one of the prevailing issues in the studied platforms was the lack of a complete representation containing all of data in a given data set. Genealogy programs like *Family Echo* or *Roots Magic* hide parts of the graphs, choosing to only represent a portion of the tree at a time. On the other hand, professional UML class diagrams editors try and represent the

full graph to the best of their capabilities, but depending on the size and number of relations, it can be impossible to create an optimal diagram. *Visual Paradigm* even recommends its users to create levels to the graph to avoid creating unreadable diagrams[1].

Proper representation of a full graph requires not only avoiding traditional graph problems, such as overlapping nodes and relations, but also to keeping the relevant information together and with defined hierarchy if applicable.

In family trees, *CleanGraph* addresses these issues by employing an algorithm that verifies placement of the nodes for collision and checks for crossing relations that could be avoided. Additionally, while placing the tree nodes, it calculates the needed space for nodes not yet placed, creating smaller clusters for each sub-family. However, it is not a perfect solution, and still has instances where fails to generate a complete graph without crossing of edges.

In the example file "Shakespear Family.ged", *CleanGraph* managed to create a representation without overlapping elements in both its layout styles (Figs. 5.1).

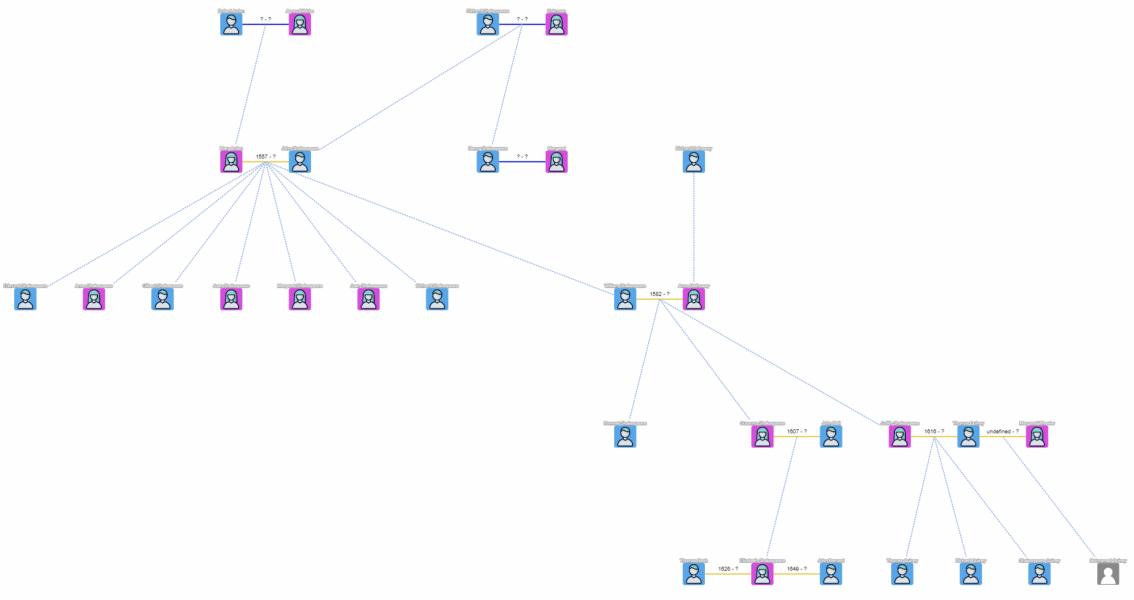


Figure 5.1: Shakespear Family.ged represented on *CleanGraph* in normal hierarchy layout.

For comparison, the graph generated by *Agelong* has three crossing edges (Fig. 5.2) that could be avoided. Due to lack of interaction on this representation, those crossing edges cannot be fixed manually.

With the same example file, *FamilyEcho* also provides a representation without crossing edges, but only due to family members being omitted (Fig. 5.3)

For UML class diagrams, *CleanGraph*'s representation is comparable in some instances but lacks the usage of hierarchy that most of the professional UML editors can provide. While the currently implemented radial layout provides a problem free representation for simple UML class diagrams (Fig. 5.4a), there are many instances where the overlapping of edges can occurs. This is

[1] https://knowhow.visual-paradigm.com/technical-support/large-project-performance/

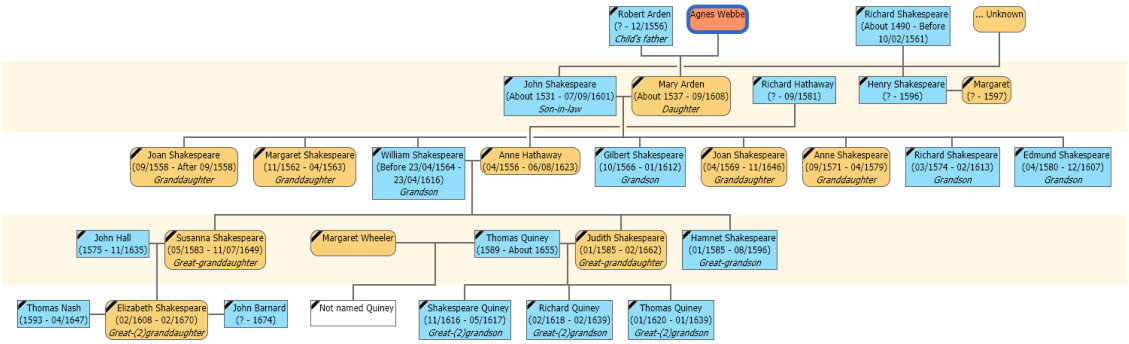


Figure 5.2: Shakespear Family.ged represented on *Agelong*.

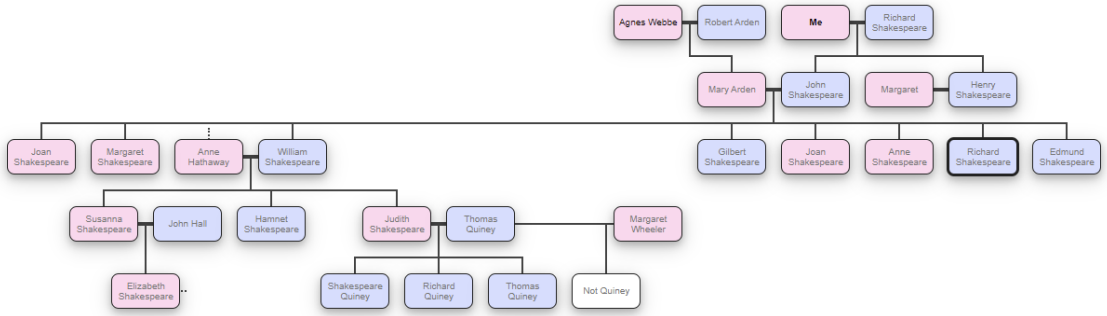
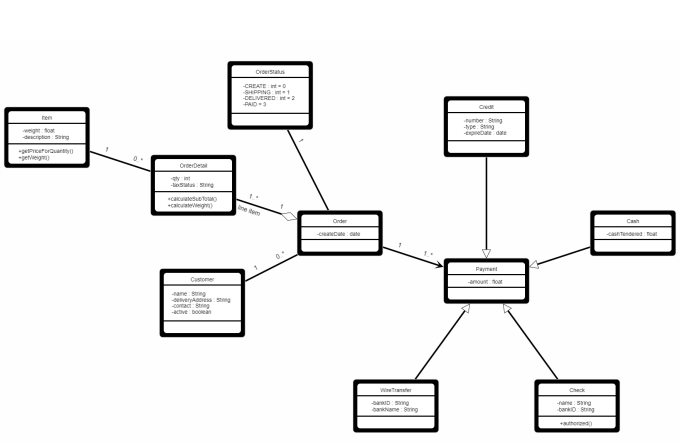
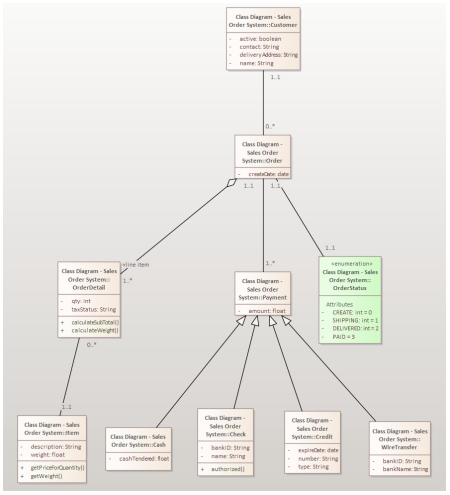


Figure 5.3: Shakespear Family.ged represented on *FamilyEcho*.

due to the radial layout algorithm not accounting for all relations when calculating the location for the classes, as well as having placement rules that are too strict and do not make use of the space available for the graph. Other programs like *Enterprise Architect* employ more complex layouts and manage to represent hierarchy accurately (Fig. 5.4b).



(a) CleanGraph.



(b) Enterprise Architect.

Figure 5.4: Representation of the Sales Order System example.

5.1.2 Unusual connections

Different programs chose different methods to handle complex or unusual connections. *CleanGraph*'s layout algorithm for family tree was created with some of the challenging representations referred to in previous chapters in mind.

When dealing with multiple relations between the same people, *CleanGraph* represents each of them, be it marriage, divorce, non-marital partnership or conception of a child, as separate lines (Fig. 5.5). The common practice amongst other genealogy platforms is only displaying the most recent of the relations.

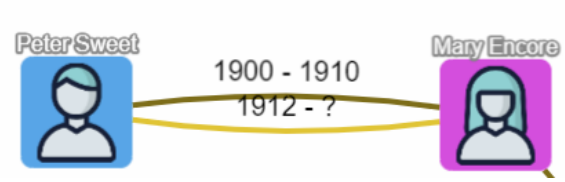


Figure 5.5: Multiple relations between the same couple in *CleanGraph*.

Simultaneous relations between more than two individuals are represented by spreading the multiple partners across the horizontal axis of the graph (Fig. 5.6). Other platforms tested either hide all but one of the marriages and require user input to select a different one, or make use of parallel relations to avoid overlapping the edges.

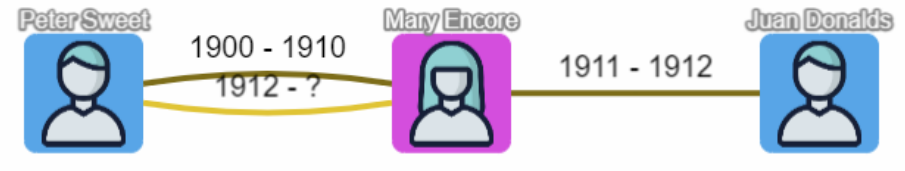


Figure 5.6: Multiple relations between different people in *CleanGraph*.

However, *CleanGraph*'s solution still has drawbacks. If there are more than two individuals relating with the same person, the relation's representation will begin to overlap.

As the representation is composed of nodes and links, it is possible to represent unusual relations such as the ones with cousins, brothers or other intergenerational pairings without resorting to node duplication (Fig. 5.7).

The placement of nodes with descendants, ancestors or both can become problematic if the tree begins spreading to where nodes are already placed. To minimise the possibility of overlapping edges, the layout algorithm attempts to place the more complex sub-families last and further away from the starting point. This is visible in the "The Kennedy Family.ged" example (Fig. 5.8), where the sub-family highlighted in blue has its descendants placed from the center outward with their complexity increasing as they are farther away. The increase in complexity is marked by the gradient from green to red, depicting the various levels of complexity.

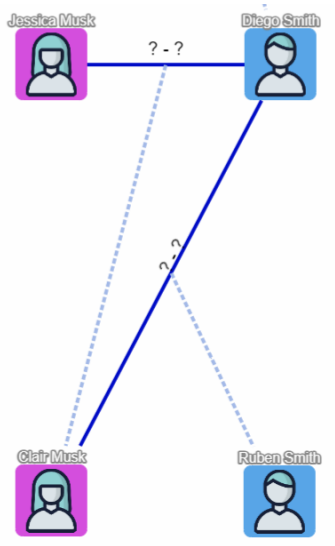


Figure 5.7: Uncommon intergenerational pairing in *CleanGraph*.

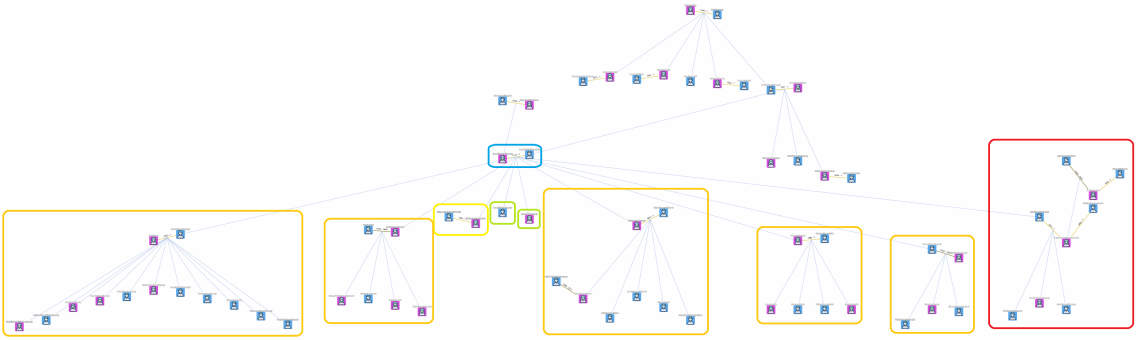


Figure 5.8: Complex sub-family handling in *CleanGraph*.

Programs such as *Agelong*, *Family Echo* and *Family.Show* chose to either hide branches of the family tree to avoid collisions or resort to using edge overlapping to allow the connections between the nodes.

5.2 User Testing

In order to evaluate the performance of the platform when being used by people with different skills, two sets of different tasks with comparable completion times where developed. The main parameters for comparison between the platforms are time to completion of each of the tasks and number of mistakes on each task and in total. If applicable, the amount by which a test subject failed a task is taken into consideration.

To provide more accurate results, the test subjects are required to pick a genealogy platform that they are used to work with. This makes the testing results better show the learning curve and ease of use of *CleanGraph*. If a user has no particular program that he has experience with, he is free to chose from many different free platforms such as *Gramps*, *Agelong*, *Family Echo*,

Family.Show or *Roots Magic*. By having people that are used to Genealogy software and people that are new to it, the ease of use of *CleanGraph* can be assessed.

The chosen tasks where:

1. **Who is the oldest/youngest person?**
2. **Was person X born earlier than person Y?**
3. **What was the birthday date of person X?**
4. **How many kids did person X have?**
5. **What was the year of X's divorce?**
6. **How many grandchildren did X have?**
7. **How many times was X married?**
8. **How many marriages have wives/husbands older than their husbands/wives?**
9. **Who is the grandfather/grandmother of X on the mother's side?**
10. **Who is the grandmother/grandfather of X on the father's side?**

For this example, X and Y in the tasks serve as placeholder people's names, and in the form that was distributed the variables were replaced with names present on the test family tree. Previous to the start of testing, the variations of these ten questions where tested in some of the Genealogy programs, in order to guarantee that the different versions, meaning the one for testing on *CleanGraph* and the one for testing on the genealogy platform chosen, would provide comparable results.

Additionally, in questions such as "How many grandchildren did X have?" the "X" chosen for each of the programs has a similar amount of grandchildren, and in questions that are reversible such as "Who is the grandfather/grandmother of X on the mothers side?", the person picked for "X" in the task for one program has similar families to the one on the other program, making it equally as easy to find regardless of platform.

Each of the tasks was designed to test a certain part of the genealogy program. Although in most genealogy programs all the task can be answered by search alone and comparing values in a text based representation, the tasks were made to be more intuitive if a proper representation is available. Questions 1, 3 and 5 are simple questions where speed relies on the quality of the search function. Questions 2 and 4 are suited for pure text search but are expected to be answered faster with a good visual representation. The remaining questions 6, 7, 8, 9 and 10 were made with the possibility of being answered purely by text but answer time is expected to be greatly improved with access to a proper visual representation of the full family tree.

By having tasks that test different parts of genealogy programs, it is expected to create a fair test to provide valid information about *Cleangraph*'s performance.

5.2.1 System Usability Scale - SUS

Considering that usability cannot be described by an absolute number, a system usability scale questionnaire was used to provide a reference of usability when comparing to existing programs. [Brooke, 1996] [Bangor et al., 2009]

The full SUS questionnaire is available in Appendix B. Following SUS rules, the questions are answered via a value ranging from a 1 to 5 that indicate how strongly a test subject disagrees or agrees with the statement. With the scores from the tests, a SUS score is calculated to then be compared to other systems.

To gather further information a set of three text based questions where created so that users could comment on what they believe are good or bad features of the platform, as well as give any recommendations that they feel would help the platform developer to meet their needs.

- What did you like the most about the program?
- What did you like the least about the program?
- What would you recommend to improve the program?

5.2.2 Results

After the creation of a form composed of twenty questions (ten per program) and a SUS survey, several genealogy communities where invited to participate in the test. These communities where contacted in several websites such as *reddit.com/r/Genealogy/*, *Gramps* forums, *Ancestry.com* forums, *Facebook* genealogy communities and *Discord* servers dedicated to Genealogy.

After sending the complete form eleven responses were obtained. In addition to the eleven forms, the communities that where reached out to also tested the platform without filling the form, which served as an endurance test for the platform The testers used five different types of Genealogy programs for their testing: *Family Echo*, *Family Tree Maker 2019*, *Gramps*, *Agelong Tree* and *Roots Magic*. Considering that the data collected was from several different programs, there was a possibility of comparing the performance of *CleanGraph* against each program specifically. However, since the sample size per application is relatively small and there is no measurable difference in performance that can be attributed to the different platforms, the final comparison is done between *CleanGraph* and the other programs as a whole.

The form created for these tests is based on the platform *Google Forms*[2] by using a HTML extractor to enable further customisation of the form. In addition to the usual questions of the form, a simple *JavaScript* script was written to capture the time each of the questions took to be answered. The first question of each of the set of ten per platform was excluded as the main goal with the first question was to start the timer for the other questions and make counting the time between answers possible. This is due to users taking time to read the instructions on the form and on each of the platforms.

[2] https://www.google.com/forms/

From all the tests, only two of the subjects did not perform better on *CleanGraph* when compared to their platform of choice (Fig. 5.1), with an average reduction of time per task of 40% across all the tests.

Table 5.1: Time spent on task and average improvement.

Tester:	Other	CleanGraph	Total	Adv
T1	0:06:56	0:04:31	0:11:27	34,86%
T2	0:04:44	0:05:13	0:09:57	-10,21%
T3	0:07:00	0:07:16	0:14:16	-3,81%
T4	0:27:22	0:07:47	0:35:09	71,56%
T5	0:14:11	0:05:55	0:20:06	58,28%
T6	0:10:03	0:07:23	0:17:26	26,53%
T7	0:12:09	0:06:04	0:18:13	50,07%
T8	0:08:51	0:03:59	0:12:50	54,99%
T9	0:15:26	0:12:11	0:27:37	21,06%
T10	0:18:15	0:11:08	0:29:23	39,00%
T11	0:14:27	0:11:00	0:25:27	23,88%
Average	0:12:40	0:07:30	0:20:10	40,85%

On average, all the tasks were completed faster on *CleanGraph*. (Fig. 5.9)

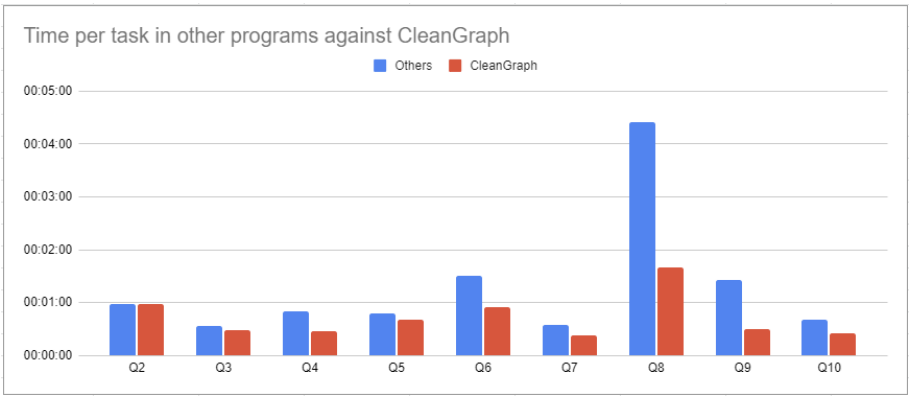


Figure 5.9: Time spent on each task compared.

The reduction in time per task is shown in Fig. 5.10. Tasks two to seven where mostly dependent on a test participant observing and clicking on UI elements, as well as using search functions and panning on graphs, making their results representative of the features implemented as well as the overall usability. By showing a reduction in time up to 46%, it was possible to conclude that *CleanGraph* was easier to use and learn than their platform of choice. Tasks eight through ten relied on the visualisation and exploration of graphs, and as such, the significant time reduction of 38% to 65% highlights the improvements in the graphical representation of *CleanGraph* when compared to other platforms.

Not only where the tasks executed faster, but there were also less mistakes made in each set of *CleanGraph* tasks, with the total of mistakes in other programs being fourteen, against seven on *CleanGraph* (Fig. 5.11), resulting in a 50% reduction in mistakes per task. Having half the

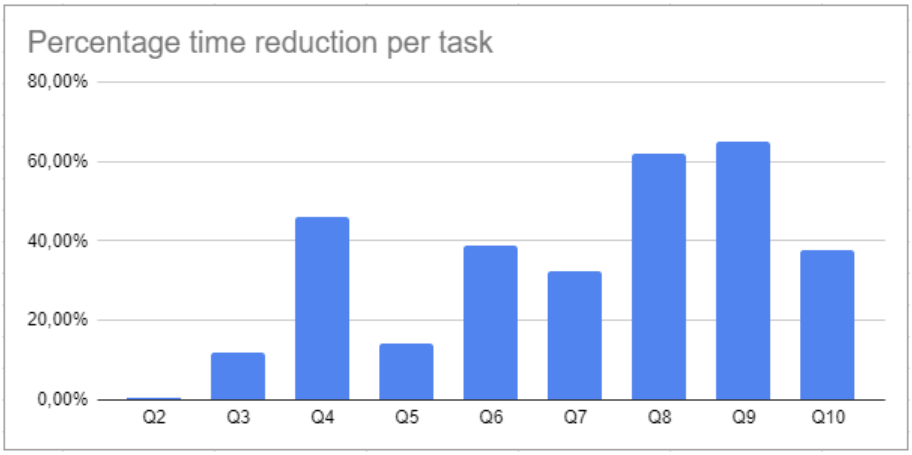


Figure 5.10: Time reduction per task.

number of mistakes than other programs in the tasks is a statement of improved usability of the developed platform when compared to others tested.

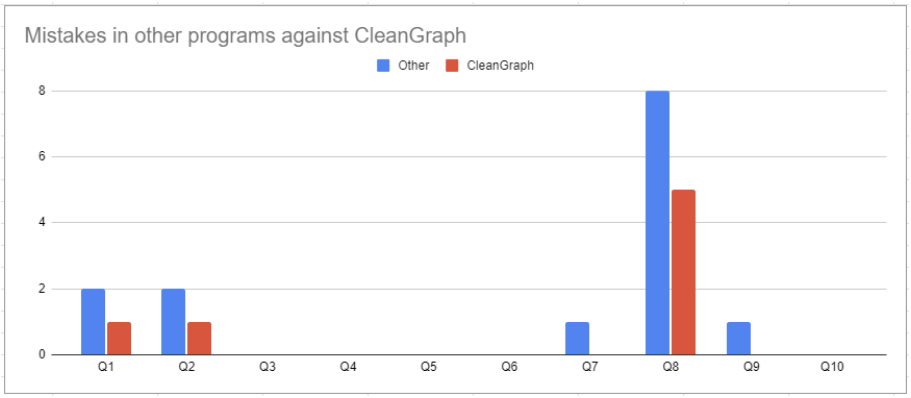


Figure 5.11: Mistakes on each task compared.

It is also relevant that in *CleanGraph* mistakes in tasks that required counting were closer to the correct answers than on other programs. Full results of the test are available in Appendix A.

From the obtained tests results it is possible to conclude that the created platform *CleanGraph* was able to produce a representation and user experience that was able to rival other Genealogy programs in speed and user accuracy. Considering that the tests where anonymous, it's not possible to pinpoint the level of proficiency of each of the test subjects. However, *CleanGraph* was a program never used by any of the test subjects, meaning that irregardless of the experience with other programs, *CleanGraph* performed very well, demonstrating its ease of use and less steep learning curve.

After users finalised the tasks for both programs they were prompted to fill the SUS questionnaire.

Following SUS guidelines, the SUS score for the program is calculated via the following equation:

$$Score = (X + Y) * 2,5$$

In this equation, X is the sum of all the scores of the odd-numbered questions minus five, and Y is twenty five minus the sum of all even-numbered. The scores from all tests are available in the Appendix B. With this data the average score of each of the SUS questionnaires was calculated, resulting in a final score of 86.

In the paper "Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale", Bangor et al. added to the SUS scale some information that would help translate it into a more descriptive scale that would allow for better judgement of the usability of an application (Fig. 5.12).

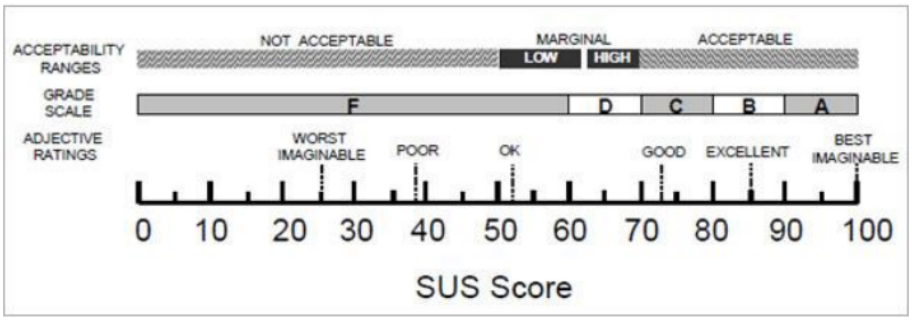


Figure 5.12: SUS Score classification from [Bangor et al., 2009].

By this metric, *CleanGraph* obtained a score that makes it acceptable and is considered excellent.

The testing also allowed test subjects to give their thoughts and opinions on the application. Positive feedback from test subjects was mostly related to the viewing capabilities and ease of use, while most negative critiques where to the lack of features compared to other more developed platforms.

According to test subjects from the family tree testing, the overall consensus was that *CleanGraph* offered an intuitive interface, the representation of the complete family tree was useful and features such as hovering people and relations were well implemented and a considered advantageous compared to other genealogy programs. The test subjects also suggested some improvements such as enlarging the auxiliary bar, adding smarter search capable of responding to complex queries and improving and improving usability by simplifying some of the UI elements as well as adding a *dark mode* to improve readability in low light environments.

Chapter 6

Conclusions and Future Work

The correct and intuitive representation of data for analysis has been a need for many people throughout the years and its necessity, as well as its capabilities, have grown with the rise of the information age and computational power. Big data collections and complex system data created the need for more apt representations that allow for user interaction and exploration.

The purpose of this work was the creation of an InfoVis platform that allowed for an enhanced representation of complex systems, with interaction and ease of use as its main goals. This concept was applied to two graph types that are distinct in content yet similar in representation: family trees and UML class diagrams.

Creating this platform required the study of already existing representations for both types of graphs. This allowed for a better understanding of the flaws of current implementations of InfoVis systems, as well as what features were expected by users for such a platform to be usable and comparable to other platforms.

Upon comparing input devices on their availability and usability for the purpose of graph manipulation and UI interaction, the choice was narrowed down to mouse cursor, keyboard and touch-screens with and without multi-touch. These input devices provided the features needed for the platform as well as providing compatibility with most devices available to the targeted audience.

From the studied graphic representations, few were applicable to both graph types needed and had possibility for added interaction. As such, the solution was to create new approaches based on existing layouts.

The created solution for family trees takes the approach of representing full family trees with defined hierarchy, avoiding hiding people by spreading the graph. Additionally, a second implemented layout uses the age of people in a family tree to place people vertically on the tree, aiding in visualisation of age differences. For UML class diagrams, the layout was formed based on both force directed and radial graphs. The usage of hierarchy was also studied for this application, and is an important consideration for future work.

For data importation, different types of file types were considered based on their capabilities,

adoption and availability of software parsers. Family trees have many file types available for storing family data, with different programs supporting different types. However there is a common file type that is considered the standard, *GEDCOM*. This format was used for both importing and exporting data across all tested platforms and as such was chosen for this implementation. Considering that UML class diagrams follow rules set by OMG when creating the standard, the file type chosen was the one created by them, *XMI*.

With the set goal of supporting a large variety of devices, the platform *CleanGraph* was created as a web-based platform. It was designed to be lightweight for users, with most of the hard processing being made on a server, allowing it to be used on mobile phones, tablet computers, desktops or any other device with a moderately recent browser. The compatibility with these different devices led to the support of input methods previously referred, such as mouse, trackpad, trackballs, keyboard, touch controls and multi-touch.

The result of this work, *CleanGraph*, is a modular platform capable of importing and representing Family Trees and UML class diagrams, as well as allowing users to explore, modify and save the graph of their choosing.

The implemented layouts for both graph types provide a complete view of the imported data while attempting to avoid common issues present in other representation platforms. The added features complement the layout by increasing interactivity and further increasing a users ability to rapidly perceive and interpret information.

Testing in a real world scenario revealed that *CleanGraph* was comparable or better than other genealogy programs in a set of tasks, especially on tasks related to visualisation and exploration of graphs. Both speed and absence of mistakes were improved when compared to existing platforms. Making an improved representation for the graph types was one of the goals of this dissertation, so the results of the testing done throughout this work provide evidence that the concept of the platform is valid and the improvements in visualisation are both feasible and desirable.

Throughout the development of *CleanGraph* various features where conceptualised and should be considered for future work:

- **Time slider** — The creation of a slider that would enable users to select a specific year in a slider bar, resulting in deceased people being greyed out and people alive at that point remaining visible. Relations would be treated in the same way, having relations highlighted depending of the state of the relation at the selected point in time, changing between marriage and divorce if applicable.

- **Images and Auto-fill images** — As of the current version of *CleanGraph*, the platform supports the display of custom images per person in a family tree, but does not allow the uploading of the images. Adding this feature would allow for further customisation of family tree graphs. In addition to user added images, *CleanGraph* could suggest images based on web searches for the people in the family tree, providing a more appealing visual experience if the family tree imported is of historical figures or celebrities.

- **Family comparison and merging** — The ability to add two families together as part of the editor for *CleanGraph* would allow users to merge two different families files that could have matching people. Additionally, this could also provide a new visualisation to compare families in a timeline, enabling users to see people in different families that were alive in the same period.

- **Editor** — With the creation of a platform for representation of graphs, the addition of an editor with file exporting capabilities is a strong consideration as it would further increase use cases for the *CleanGraph* platform. This would allow the platform to be more complete, capable of creation, editing and representation of these diagrams, removing the need of extraneous platforms. With an editor implemented, exportation of *GEDCOM* and *XMI* should be considered as well, making *CleanGraph* more competitive with other graph representation programs.

- **UML class diagram radial highlight** — One of the features that was considered for UML class diagram representation to aid in visualisation of complex diagrams was the addition of a radial highlight for the UML classes. When a user selected an UML class and activated the feature, other UML classes would surround the selected one, forming circular rings around it. The distance between these classes and the selected class would equate to how related these other classes are to the center one.

- **UML class diagram overview** — As UML class diagrams can become very complex, creating a representation focused on grouping classes based on their content and context could be feature that aids interpretation of the systems. This could be achieved via the a force-directed graph algorithm.

Furthermore, considering that data conversion from the chosen filetypes (*GEDCOM* and *XMI*) to the *CleanGraph* representations was created for this project, there are still improvements that need to be made in both the parsers and converters. The *GEDCOM* parser chosen has some issues in some specific files that can lead to the layout algorithms not being able to perform properly. By creating a new parser, these issues could be addressed and would allow for better customizability of the JSON object produced. Additionally, the *XMI* parser was created from the ground up to support the UML 2.1 *XMI* standard, but still lacks the capability to process some of data of the filetype.

Although the layouts implemented proved effective in the tasks there is still improvements required for both the representation of family trees and UML class diagrams. The basic layout for family trees, although reliable in family trees with smaller subfamilies of 3-5 children with simple relations, struggles to represent larger trees with more complex cases without overlapping relations. The current implementation makes use of a grid-like system to place nodes which is not ideal for every family tree and can create instances where space is left unoccupied needlessly. This could be improved by allowing the collision and positioning algorithms to use more precise measurements when placing people, at the cost of performance. The layout where year of birth

is represented by vertical position in the family tree would also be improved by addition of the improved collision checking. Additionally, more of the information from the *GEDCOM* file type should be processed and displayed properly, including better processing for events and special types of notes.

For UML class diagram, the base representation is not flexible enough for cases where many classes exist. To improve the layout, the positioning algorithm should be more aware of free space when calculating the angle in which the graph can spread. Furthermore, the algorithm should make use of the distance between the classes to avoid collisions and overlapping. The layout should also be made to use hierarchy, positioning classes with generalisation relationships in mind.

All in all, *CleanGraph* shows a new direction in Family Tree and UML class diagram representation, bringing new features that proved useful in a real-life user testing. It tackles readability, ease of use and clarity all in a new platform, improving speed and reducing misinterpretation of these type of graphs. However, more work is needed to improve this platform further, in order to achieve a new, complete program that can rival the established ones in these areas.

Bibliography

[Allingham, 2020] Allingham, D. N. (2020). Gramps - Genealogical Research Software. Retrieved from https://gramps-project.org/, visited on 2020-12-04.

[Alsop, 2020] Alsop, T. (2020). Tablets, laptops & PCs sales forecast 2023 | Statista. Retrieved from https://www.statista.com/statistics/272595/global-shipments-forecast-for-tablets-laptops-and-desktop-pcs/, visited on 2021-01-02.

[Bangor et al., 2009] Bangor, A., Kortum, P., and Miller, J. (2009). Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of usability studies*, 4(3):114–123.

[Basheri et al., 2012] Basheri, M., Burd, L., and Baghaei, N. (2012). A multi-touch interface for enhancing collaborative uml diagramming. In *Proceedings of the 24th Australian Computer-Human Interaction Conference, 26-30 November 2012, Melbourne, Australia*, OzCHI '12, page 30–33, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2414536.2414541.

[Basheri et al., 2013] Basheri, M., Burd, L., Munro, M., and Baghaei, N. (2013). Collaborative learning skills in multi-touch tables for uml software design. *International Journal of Advanced Computer Science and Applications*, 4:60–66. DOI: 10.14569/IJACSA.2013.040311.

[Bellis, 2020] Bellis, M. (2020). The History of the Computer Keyboard. Retrieved from https://www.thoughtco.com/history-of-the-computer-keyboard-1991402, visited on 2020-12-15.

[Bezerianos et al., 2010] Bezerianos, A., Dragicevic, P., Fekete, J. D., Bae, J., and Watson, B. (2010). GeneaQuilts: A system for exploring large genealogies. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1073–1081. DOI: 10.1109/TVCG.2010.159.

[Bostock and Heer, 2009] Bostock, M. and Heer, J. (2009). Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128.

[Brooke, 1996] Brooke, J. (1996). *"SUS-A quick and dirty usability scale." Usability evaluation in industry*. CRC Press. ISBN: 9780748404605.

[Burton, 2020] Burton, G. (2020). Graph View - Gramps Addon. Retrieved from https://gramps-project.org/wiki/index.php/Graph_View, visited on 2021-01-17.

[Chen et al., 2009] Chen, M., Laramee, R. S., Ebert, D., Hagen, H., van Liere, R., Ma, K. L., Ribarsky, W., Scheuermann, G., and Silver, D. (2009). Data, Information, and Knowledge in Visualization. *IEEE Computer Graphics and Applications*, 29(1):12–19. DOI: 10.1109/MCG.2009.6.

[Cook et al., 2017] Cook, S., Bock, C., Rivett, P., Rutt, T., Seidewitz, E., Selic, B., and Tolbert, D. (2017). Unified modeling language (UML) version 2.5.1. Standard, Object Management Group (OMG).

[Ellson et al., 2002] Ellson, J., Gansner, E., Koutsofios, L., North, S. C., and Woodhull, G. (2002). Graphviz - Open source graph drawing tools. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 23-26 September 2001, Vienna, Austria*, volume 2265 LNCS, pages 483–484. Springer Verlag. DOI: 10.1007/3-540-45848-4_57.

[Everett, 1980] Everett, R. R. (1980). WHIRLWIND. In *A History of Computing in the Twentieth Century*, pages 365–384. Elsevier. DOI: 10.1016/b978-0-12-491650-0.50025-3.

[Franz et al., 2016] Franz, M., Lopes, C. T., Huck, G., Dong, Y., Sumer, O., and Bader, G. D. (2016). Cytoscape. js: a graph theory library for visualisation and analysis. *Bioinformatics*, 32(2):309–311.

[Friendly and Wainer, 2021] Friendly, M. and Wainer, H. (2021). *A History of Data Visualization and Graphic Communication*. Harvard University Press. DOI: 10.4159/9780674259034.

[Fu et al., 2013] Fu, B., Noy, N. F., and Storey, M.-A. (2013). Indented tree or graph? a usability study of ontology visualization techniques in the context of class mapping evaluation. In *Proceedings of the 12th International Semantic Web Conference - Part I, 21-25 October 2013, Sydney, Australia*, ISWC '13, page 117–134, Berlin, Heidelberg. Springer-Verlag. DOI: 10.1007/978-3-642-41335-3_8.

[Genery Software, 2021] Genery Software (2021). Agelong. https://genery.com/, visited on 2021-06-02.

[Greenspan, 2015] Greenspan, G. (2015). Gideon Greenspan. http://www.gidgreen.com/, visited on 2021-06-20.

[Heer and Boyd, 2005] Heer, J. and Boyd, D. (2005). Vizster: Visualizing online social networks. In *Proceedings of IEEE Symposium on Information Visualization (InfoVis '05), 23-25 October 2005, Minneapolis, MN, USA*, pages 32–39. DOI: 10.1109/INFVIS.2005.1532126.

[Heer et al., 2005] Heer, J., Card, S. K., and Landay, J. A. (2005). Prefuse: a toolkit for interactive information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems, 2-7 April 2005, Portland, Oregan*, pages 421–430.

[Holst, 2021] Holst, A. (2021). Top obstacles to mass adoption of VR technologies 2019 | Statista. Retrieved from https://www.statista.com/statistics/1098566/obstacles-to-mass-adoption-of-vr-technologies/, visited on 2021-01-08.

[Huang et al., 2017] Huang, Y. J., Fujiwara, T., Lin, Y. X., Lin, W. C., and Ma, K. L. (2017). A gesture system for graph visualization in virtual reality environments. In *Proceedings of IEEE Pacific Visualization Symposium, 18-21 April 2017, Seoul, South Korea*, pages 41–45, Seoul, South Korea. IEEE Computer Society. DOI: 10.1109/PACIFICVIS.2017.8031577.

[Johnson, 1965] Johnson, E. A. (1965). Touch display–a novel input/output device for computers. *Electronics Letters*, 1(8):219–220.

[Johnson, 1967] Johnson, E. A. (1967). Touch displays: A programmed man-machine interface. *Ergonomics*, 10(2):271–277. DOI: 10.1080/00140136708930868.

[Kaidi, 2000] Kaidi, Z. (2000). Data visualization. *Retrieved*, 8(22):2010.

[Keller et al., 2011] Keller, K., Reddy, P., and Sachdeva, S. (2011). Family tree visualization. *Berkeley, CA, USA*.

[Kobourov, 2012] Kobourov, S. G. (2012). Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv:1201.3011*.

[Kosara et al., 2003] Kosara, R., Hauser, H., and Gresh, D. L. (2003). An interaction view on information visualization. In *Eurographics (State of the Art Reports), 1-5 September 2003, Granada, Spain*.

[Lee et al., 2012] Lee, B., Isenberg, P., Riche, N. H., and Carpendale, S. (2012). Beyond mouse and keyboard: Expanding design considerations for information visualization interactions. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2689–2698. DOI: 10.1109/TVCG.2012.204.

[Lee et al., 2018] Lee, B., Srinivasan, A., Stasko, J., Tory, M., and Setlur, V. (2018). Multimodal interaction for data visualization. In *Proceedings of the Workshop on Advanced Visual Interfaces AVI, May 2018, Grosseto, Italy*, pages 1–3, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3206505.3206602.

[Leeuw and Michailidis, 2000] Leeuw, J. D. and Michailidis, G. (2000). Graph layout techniques and multidimensional data analysis. *Lecture Notes-Monograph Series*, 35:219–248.

[Munafo et al., 2017] Munafo, J., Diedrick, M., and Stoffregen, T. A. (2017). The virtual reality head-mounted display Oculus Rift induces motion sickness and is sexist in its effects. *Experimental Brain Research*, 235(3):889–901. DOI: 10.1007/s00221-016-4846-7.

[Nakagawa and Uwano, 2011] Nakagawa, T. and Uwano, H. (2011). Usability evaluation for software keyboard on high-performance mobile devices. In *Proceedings of Communications*

in Computer and Information Science, 9-14 July 2011, Orlando, Florida, USA, volume 173 CCIS, pages 181–185. Springer, Berlin, Heidelberg. DOI: 10.1007/978-3-642-22098-2_37.

[Nguyen et al., 2020] Nguyen, W., Lake, H., and Santiago, A. (2020). VR Motion Sickness Statistics | Age, Gender, Experience and More | VR Heaven. Retrieved from https://vrheaven.io/vr-motion-sickness-statistics/, visited on 2021-01-09.

[O'madadhain et al., 2005] O'madadhain, J., Fisher, D., Smyth, P., White, S., and Boey, Y.-B. (2005). Journal of Statistical Software Analysis and Visualization of Network Data using JUNG. *Journal of Statistical Software*, 10(2):1–35.

[Pang, 2002] Pang, A. S.-K. (2002). The making of the mouse. *American Heritage of Invention and Technology*, 17(3):48–54.

[Purchase, 2014] Purchase, H. C. (2014). Twelve years of diagrams research. *Journal of Visual Languages and Computing*, 25(2):57–75. DOI: 10.1016/j.jvlc.2013.11.004.

[Riaz and Ali, 2011] Riaz, F. and Ali, K. M. (2011). Applications of graph theory in computer science. In *Proceedings of the Third International Conference on Computational Intelligence, Communication Systems and Networks, 26-28 July 2011, Bali, Indonesia*, pages 142–145. DOI: 10.1109/CICSyN.2011.40.

[Richards Franklin, 2012] Richards Franklin, Musser Milton, A. J. (2012). GEDCOM X. Retrieved from http://www.gedcomx.org/, visited on 2021-01-10.

[Santos et al., 2013] Santos, J. M., Santos, B. S., Dias, P., Silva, S., and Ferreira, C. (2013). Extending the h-tree layout pedigree: An evaluation. In *2013 17th International Conference on Information Visualisation*, pages 422–427.

[Schmidt and Churchill, 2012] Schmidt, A. and Churchill, E. (2012). Interaction beyond the keyboard. *Computer*, 45(4):21–24. DOI: 10.1109/MC.2012.137.

[Search, 2021] Search, F. (2021). GEDCOM - FamilySearch Developer Center — FamilySearch.org. Retrieved from https://www.familysearch.org/developers/docs/guides/gedcom, visited on 2021-02-11.

[Snehi, 2006] Snehi, J. (2006). *Computer peripherals and interfacing*. Firewall Media.

[Sorapure, 2019] Sorapure, M. (2019). Text, Image, Data, Interaction: Understanding Information Visualization. *Computers and Composition*, 54:102519. DOI: 10.1016/j.compcom.2019.102519.

[Sparks, 2020] Sparks, G. (2020). UML modeling tools for Business, Software, Systems and Architecture. Retrieved from https://sparxsystems.com/, visited on 2020-12-20.

[Svinterikou and Theodoulidis, 1999] Svinterikou, M. and Theodoulidis, B. (1999). Tuml: A method for modelling temporal information systems. In *Proceedings of International Conference on Advanced Information Systems Engineering, 14-18 June 1999, Berlin, Heidelberg*, pages 456–461. Springer.

[team, 2020] team, V. P. (2020). What is Unified Modeling Language (UML)? Retrieved from https://www.visual-paradigm.com/guide/uml-unified-modeling-language, visited on 2021-02-11.

[Victorelli et al., 2020] Victorelli, E. Z., Dos Reis, J. C., Hornung, H., and Prado, A. B. (2020). Understanding human-data interaction: Literature review and recommendations for design. *International Journal of Human Computer Studies*, 134:13–32. DOI: 10.1016/j.ijhcs.2019.09.004.

[Walny et al., 2012] Walny, J., Lee, B., Johns, P., Henry Riche, N., and Carpendale, S. (2012). Understanding pen and touch interaction for data exploration on interactive whiteboards. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2779–2788. DOI: 10.1109/TVCG.2012.275.

[Ware, 2012] Ware, C. (2012). *Information Visualization: Perception for Design*. Morgan Kaufmann Series in Interactive Technologies. Morgan Kaufmann, Amsterdam, 3 edition.

[Yi et al., 2007] Yi, J. S., Kang, Y. A., Stasko, J. T., and Jacko, J. A. (2007). Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231. DOI: 10.1109/TVCG.2007.70515.

Appendix A

Test answers

As a first step please write the name of the program you will be using.	Who is the youngest person in the family?	Is Charles Joseph Burke older than John Vernou Bouvier III?	What is the birthday of Virginia Joan Bennett? (Format it like dd/mm/yyyy)	How many kids did John Fitzgerald Kennedy have?	What was the year of Janet Norton Lee's divorce?	How many grandkids did Bridget Murphy have?
familyecho.com	Rory Kennedy	Yes	02/09/1936	3	1940	4
Family Echo	Rory Elizabeth Katherine	No	02/09/1936	3	1940	4
Family Tree Maker 2019	Rory Elizabeth Katherine	No, John Vernou Bouvier	02/09/1936	3	1940	4
Gramps	Patrick Bouvier Kennedy	No	02/09/1936	3	1940	4
Agelong Tree	Rory Elizabeth Katherine	No	02-09-1936	3	1940	4
Agelong Tree	Rory Elizabeth Katherine	No.	02/09/1936	3	1940	4
Family Echo	Rory Elizabeth Katherine	No	02/09/1936	3	1940	4
familyecho	Rory Kennedy	no	02/09/1936	3	1940	4
RootsMagic	Rory Elizabeth Kennedy	Yes	02/09/1936	3	1940	4
Family Echo	Patrick Kennedy	No	02/09/1936	3	1940	4
Agelong	Rory Elizabeth Katherine	No	02/09/1936	3	1940	4

Figure A.1: First part of user answers.

How many times was Janet Norton Lee married?	How many marriages have husbands that are older than their wives?	Who is the grandfather of Rose Marie Kennedy on the mothers side?	Who is the grandmother of Margaret Louise Kennedy on the fathers side?	Who is the oldest person in the family?	Is Mary Josephine Hannon older than Humphrey Charles Mahoney?	What is the birthday of Patricia Helen Kennedy? (Format it like dd/mm/yyyy)
2	10	John Fitzgerald	Bridget Murphy	Bridget Murphy	No	06/05/1924
1	9	John Francis Fitzgerald	Bridget Murphy	Bridget Murphy	No dates for Mary Josep	06/05/1924
2	Unknown	John Francis Fitzgerald	Bridget Murphy	Bridget Murphy	No birth info is listed for M	06/05/1924
2	5	John Francis Fitzgerald	Bridget Murphy	Bridget Murphy	No	06/05/1924
2	9	John Francis Fitzgerald	Bridget Murphy	Bridget Murphy	No	06-05-1924
2	14	John Francis Fitzgerald	Bridget Murphy	Bridget Murphy	No.	1924/05/06
2	8	John Francis Fitzgerald	Bridget Murphy	Bridget Murphy	No	06/05/1924
2	11	John Fitzgerald	Bridget Murphy	Bridget Murphy	no	06/05/1924
2	16	John Francis "Honey Fitz	Bridget Murphy	Rory Elizabeth Katherine	Yes	06/05/1924
2	9	John Francis Fitzgerald	Bridget Murphy	Bridget Murphy	no	6/5/1924
2	12	Rose Marie Kennedy	Bridget Murphy	Bridget Murphy	No	06/05/1924

Figure A.2: Second part of user answers.

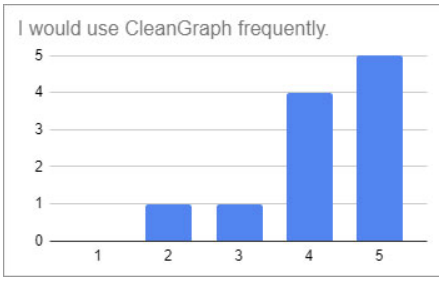
How many kids did Eunice Mary Kennedy have?	What was the year of Arnold Alois Schwarzenegger's divorce?	How many grandkids did Janet Norton Lee have?	How many times was Jacqueline Lee Kennedy Bouvier married?	How many marriages have wives that are older than their husbands?	Who is the grandmother of Rose Marie Kennedy on the mothers side?	Who is the grandfather of Margaret Louise Kennedy on the fathers side?
5	2011	3	2	7	Mary Josephine Hannon	Patrick Kennedy
5	2011	3	2	6	Mary Josephine Hannon	Patrick Kennedy
5	2011	3	2	3	Mary Josephine Hannon	Patrick Kennedy
5	2011	3	2	6	Mary Josephine Hannon	Patrick Kennedy
5	2011	3	2	5	Mary Josephine Hannon	Patrick Kennedy
5	2011	3	2	6	Mary Josephine Hannon	Patrick Kennedy
5	2011	3	2	5	Mary Josephine Hannon	Patrick Kennedy
5	2011	3	2	6	Mary Josephine Hannon	Patrick Kennedy
5	2011	3	2	6	Mary Josephine Hannon	Patrick Kennedy
5	2011	3	2	5	Mary Josephine Hannon	Patrick Kennedy
5	2011	3	2	6	Mary Josephine Hannon	Patrick Kennedy

Figure A.3: Third part of user answers.

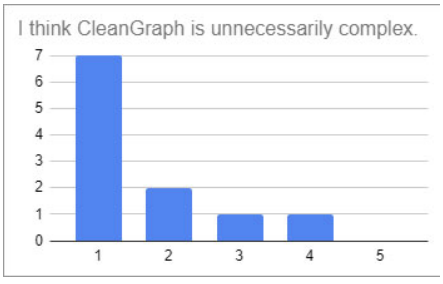
Appendix B

SUS Questionnaire and scores

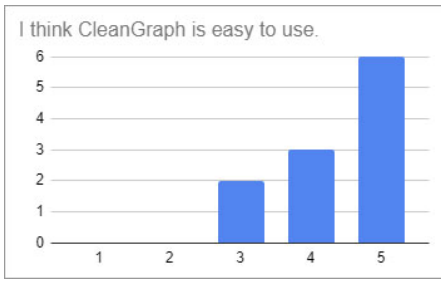
- I would use CleanGraph frequently.
- I think CleanGraph is unnecessarily complex.
- I think CleanGraph is easy to use.
- I think I would require assistance from someone with technical knowledge of CleanGraph.
- I think many of the features of CleanGraph are well integrated.
- I think CleanGraph has many inconsistencies.
- I think most people will learn how to use CleanGraph quickly.
- I think CleanGraph is hard to use and learn.
- I felt confident using CleanGraph.
- I had to learn many things before using CleanGraph.



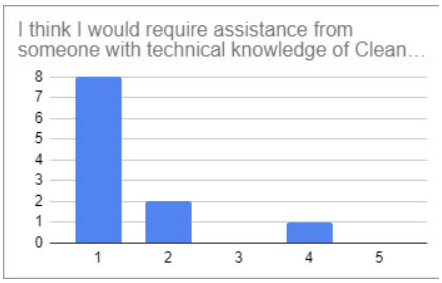
(a) Test 1



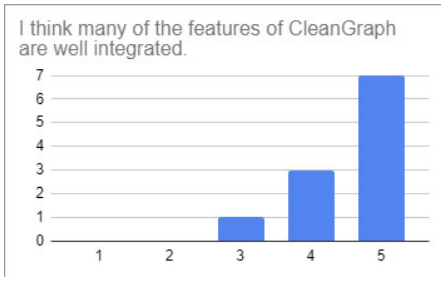
(b) Test 2



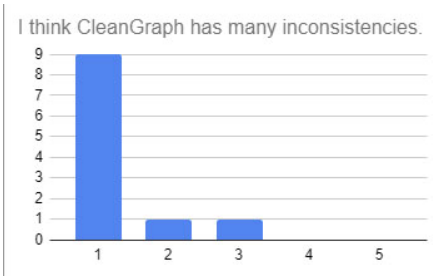
(a) Test 3



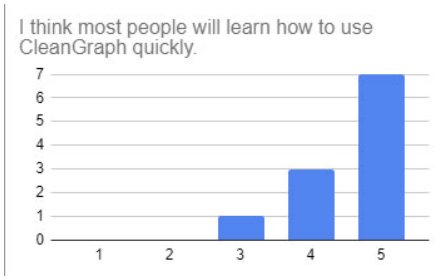
(b) Test 4



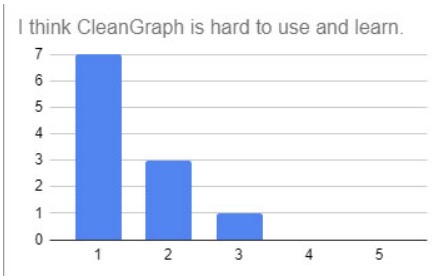
(a) Test 5



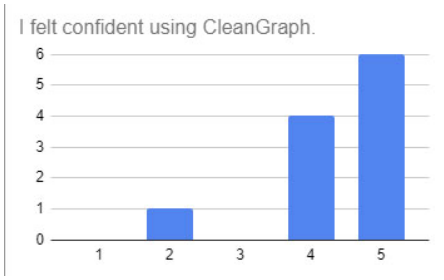
(b) Test 6



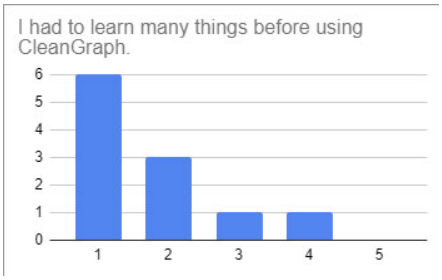
(a) Test 7



(b) Test 8



(a) Test 9



(b) Test 10

Appendix C

UML placing logic

```
1
2  function spreadRadius (node, distance, angleFrom, angleTo, centerX, centerY,
       visited) {
3      let nodes = getUnvisitedConnectedNodes(node, visited);
4      let elementNumb = nodes.length;
5      let toSpread = [];
6      let spacing = (angleTo-angleFrom) / elementNumb;
7
8      nodes.forEach((child, idx, children) => {
9          if (!visited.has(child.data("id"))){
10             let xPos;
11             let yPos;
12             if(elementNumb === 1){
13                 xPos = Math.round(centerX + distance * Math.cos((angleFrom + (
                       angleTo - angleFrom)/2) ));
14                 yPos = Math.round(centerY + distance * Math.sin((angleFrom + (
                       angleTo - angleFrom)/2) ));
15             } else if ((angleTo - angleFrom) === (2*Math.PI)){
16                 xPos = Math.round(centerX + distance * Math.cos((angleFrom + (
                       spacing * idx)) ));
17                 yPos = Math.round(centerY + distance * Math.sin((angleFrom + (
                       spacing * idx)) ));
18             } else {
19                 spacing = (angleTo-angleFrom) / (elementNumb - 1);
20                 xPos = Math.round(centerX + distance * Math.cos((angleFrom + (
                       spacing * idx)) ));
21                 yPos = Math.round(centerY + distance * Math.sin((angleFrom + (
                       spacing * idx)) ));
22             }
23             child.position({x: xPos, y: yPos});
24             let angle = Math.atan((yPos-centerY)/(xPos-centerX));
25             if (xPos < centerX)
26                 angle += Math.PI;
```

```
27              let unvisitedChildren = getUnvisitedConnectedNodes(child, visited);
28              if (unvisitedChildren.length > 0) {
29                  toSpread.push({
30                      node: child,
31                      distance: distance,
32                      angleFrom: angle - (Math.PI/4),
33                      angleTo: angle + (Math.PI/4),
34                      centerX: xPos,
35                      centerY: yPos
36                  })
37              }
38              visited.set(child.data("id"), {x: xPos, y: yPos});
39          }
40      })
41      toSpread.forEach(child => {
42          spreadRadius (child.node, child.distance, child.angleFrom, child.angleTo,
                child.centerX, child.centerY, visited);
43      })
44  }
```